AB-tree: Index for Concurrent Random Sampling and Updates

Zhuoyue Zhao[*]
University at Buffalo
zzhao35@buffalo.edu

Dong Xie
The Pennsylvania State University
dongx@psu.edu

Feifei Li
Alibaba
lifeifei@alibaba-inc.com

ABSTRACT

There has been an increasing demand for real-time data analytics. Approximate Query Processing (AQP) is a popular option for that because it can use random sampling to trade some accuracy for lower query latency. However, the state-of-the-art AQP system either relies on scan-based sampling algorithms to draw samples, which can still incur a non-trivial cost of table scan, or creates samples of the database in a preprocessing step, which are hard to update. The alternative is to use the aggregate B-tree indexes to support both random sampling and updates in database with logarithmic time. However, to the best of our knowledge, it is unknown how to design an aggregate B-tree to support highly concurrent random sampling and updates, due to the difficulty of maintaining the aggregate weights correctly and efficiently with concurrency. In this work, we identify the key challenges to achieve high concurrency and present AB-tree, an index for highly concurrent random sampling and update operations. We also conduct extensive experiments to show its efficiency and efficacy in a variety of workloads.

PVLDB Reference Format:
Zhuoyue Zhao, Dong Xie, and Feifei Li. AB-tree: Index for Concurrent Random Sampling and Updates. PVLDB, 15(9): 1835 - 1847, 2022.
doi:10.14778/3538598.3538606

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zzy7896321/abtree_public.

1 INTRODUCTION

Approximate Query Processing (AQP) has become very appealing for real-time data analytics, especially on very large databases where table scans or index range scans are too slow to complete. It provides a unique trade-off between result accuracy and query latency for analytical queries. A major approach of AQP is to estimate query results on top of random samples drawn from the base tables in a database. There have been a number of works on using different random distributions to produce accurate approximate answers for aggregation queries [3, 6, 13, 19, 25], as well as designing algorithms for drawing random samples from the results of multi-table joins [2, 5, 31]. In fact, one can answer simple aggregation queries approximately with theoretical guarantees in a sub-linear or even constant time relative to the data size [6, 9]. As a result, AQP can often save hours or even days of query time by only aggregating a slice out of terabytes or petabytes of data.

All AQP techniques above treat the sample drawing mechanism as a black box, assuming they can be efficiently implemented using SQL or by extending existing DBMS with minor modifications. It, however, is not true if we were to perform AQP for real-time data analytics where concurrent data updates exist. Existing `TABLESAMPLE` operator supported by the SQL in all major DBMS implementation requires either linear scan (Bernoulli sampling) or relaxed statistical guarantees (system or block sampling). Note that the latter is unacceptable for AQP since it invalidates the error guarantees. Most other sampling techniques such as stratified sampling, universe sampling and distinct sampling [13] require full table or index range scans over base tables. This diminishes the benefit of AQP as scanning can be a dominating factor of the query cost, especially when the data are stored in external storage. Therefore, many AQP systems [3, 6, 25] opt to draw samples *offline* and only update them once in a while, making real-time analytics infeasible.

In order to support real-time data analytics with AQP, we need the ability of drawing random samples *online* without scanning the base tables. Luckily, it is well understood that aggregate B-trees can be used for drawing random samples online in logarithmic time relative to data size per random sample [5, 24, 31]. An aggregate B-tree is a B-tree index whose internal nodes are annotated with some aggregated weights of their corresponding subtrees. However, to the best of our knowledge, it is unknown how to make correct and efficient concurrent updates in aggregate B-trees without latching the entire structure. That means, an AQP system that uses aggregate B-tree may perform reasonably good over rarely updated data, but its performance will drop rapidly as more data updates are involved.

The crux of this problem is that every single data update will involve changes to the aggregated weights on all the internal nodes along a tree path. It always leads to very high contention on higher level nodes (especially the root). Besides, it may expose an inconsistent view of weights to concurrent readers, leading to biased samples. We will show in Section 2.2 that the second issue may be alleviated by imposing strict ordering of weight updates along a path, but could still harm concurrency significantly when Structural Modification Operations (SMO, i.e., page split/merge) and sampling operations are interleaved. Unfortunately, the conventional wisdom of the existing high-performance concurrent B-tree designs [4, 15, 16, 18, 20, 26, 27] (where most contentions happen around the leaf level, and SMOs can be implemented as a few atomic steps that may interleave with other operations) is no longer applicable as their assumptions are completely the opposite. In addition, as a practical matter, multi-version concurrency control (MVCC), which is often used by modern DBMS to improve concurrency, can negatively impact the random sampling efficiency of aggregate B-trees under concurrent updates due to sample rejections induced by the invisible versions.

[*]Part of the work was done while the author was a PhD student at University of Utah.

Proceedings of the VLDB Endowment, Vol. 15, No. 9 ISSN 2150-8097.
doi:10.14778/3538598.3538606

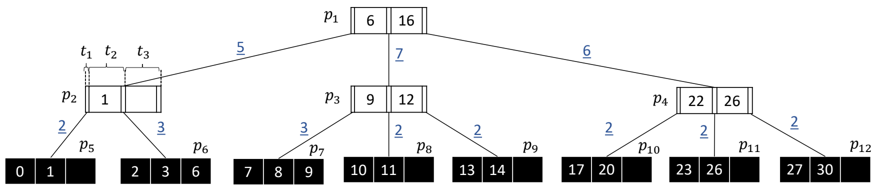


Figure 1: An example aggregate B-tree with height 3 and a fan-out of 3.

In this paper, we present AB-tree, a concurrent aggregate B-tree based on the B-link tree [16]. It can support correct and highly concurrent updates and sampling queries. Besides, it features a latch-free multi-version weight store that stores a history of versioned weight updates to further improve the sampling efficiency by avoiding some sample rejections. To the best of our knowledge, this is the first work to design a highly concurrent aggregate B-tree. Our major technical contributions lie in the following aspects:

- We identify the main design problems in building an efficient concurrent aggregate B-tree and discuss how they deviate from those of the traditional concurrent B-tree designs.
- We propose several general design principles for building an efficient concurrent aggregate B-tree.
- Incorporating our general design principles, we design and implement AB-tree, an efficient concurrent aggregate B-tree based on the B-link tree to support highly concurrent updates and random sampling.
- We conduct comprehensive evaluations to show the efficiency of our method in a real-world DBMS (PostgreSQL).
- We discuss the possibility of generalizing our design to support other index structures and sampling operations (e.g., independent range sampling).

The rest of this paper is organized as follows. We introduce the necessary background, the challenges and the requirements in designing a concurrent B-tree in Section 2. We describe the general design principles of AB-tree in Section 3 and the practical implementation details of it in PostgreSQL in Section 4. In particular, we discuss how to reduce sample rejections under MVCC in Section 4.4. We showcase AB-tree's performance advantages to the best available baseline using a simple synthetic dataset and the more realistic TPC-H dataset in Section 5. Finally, we review the related works and discuss the possibility of generalizing our design in Section 6. We conclude this paper in Section 7.

2 PRELIMINARIES

The goal of this paper is to design an aggregate B-tree that can sustain a high rate of concurrent update (insert/delete) and random sampling operations. Sampling operations draw weighted random samples out of all leaf tuples in the B-tree, where the probability of sampling a leaf tuple t is proportional to some specified weight w_t. As in any concurrent B-tree, the effects of updates should be serializable. Sampling in an aggregate B-tree should only return valid samples at certain point in the serialized update history. To formalize that, our *basic correctness requirement* for a sampling operation is that the weighted random samples withdrawn should be respective to some non-empty set of leaf tuples resulted from a prefix of the equivalent serial schedule. For instance, suppose the equivalent serial schedule of 3 inserts is inserting tuples 1, 2, 3 with equal weights. Then, a concurrent random sampling operation may draw a random sample from one of these sets $\{1\}, \{1, 2\}, \{1, 2, 3\}$ uniformly, but not any other sets (e.g., $\{1, 3\}$) and/or with a different probability distribution. In a typical AQP usage, one may perform many independent sampling operations on the same snapshot of the database as newer versions get inserted into the index without removing the old versions that are still visible under the snapshot. Suppose the set of visible tuples under the snapshot is S. Then the basic correctness requirement would guarantee that for each sample s drawn from the index, $\forall s_0 \in S, Pr\{s = s_0\} \propto w_{s_0}$, because S can never change for the same snapshot in any allowable serial schedule. Hence, one may use rejection sampling to draw as many independent samples from the same distribution as it wants even though the index may return samples not in S. The approach would not work for an index that does not satisfy the basic correctness requirement. For instance in our previous example, suppose a sampler runs on a snapshot where $S = \{1, 2\}$, but the sampler returns a uniform sample from the set $\{1, 3\}$. Then $Pr(s = 1) = 1$ and $Pr(s = 2) = 0$ in this case, which is no longer uniform as it is supposed to be.

To the best of our knowledge, there is no existing design that can satisfy our basic correctness requirement with high concurrency. A simple and best-available baseline is to exclusively latch all the pages along the search path of an update and only release the latches when all the weights are updated. However, this effectively blocks all other operations since the root is always in contention.

2.1 The Structure of Aggregate B-trees

An aggregate B-tree (see Figure 1 for an example) consists of a number of leaf pages and a number of internal pages, organized in h levels (also denoted as the height of the tree). All tuples in a level are sorted by the their indexed keys and, without loss of generality, we assume all keys are unique. Level 0 is the leaf level and stores all *leaf tuples* (marked black in Figure 1). A leaf tuple t corresponds to a unique heap tuple and contains a key k_t and a pointer to that heap tuple. The internal pages start from level 1 until level $h - 1$. An index tuple t on level l contains a page pointer c_t to a page on level $l - 1$, which is the root of its corresponding subtree $\mathcal{T}_t$. It is also associated with a key k_t that is a strict lower bound of the keys of all leaf tuples in $\mathcal{T}_t$. Note that there is no need to store k_t for the first tuple in an internal page as it is implicitly the same as the key associated with the tuple that points to the page.

2.2 Random Sampling in Aggregate B-trees

To perform random sampling in an aggregate B-tree, we associate each leaf tuple t with a weight $w_t = f(t)$, where $f(t) \propto Pr(t)$ in the desired distribution. For instance, $f(t) = 1$ if we want to perform uniform sampling, or $f(t) = t_M$ for some measure column M if we want to perform measure-biased sampling to optimize sample variance [6]. The value of w_t does not need to be stored explicitly in the leaf tuple unless it is very expensive to evaluate or involve a non-deterministic weight function f. We also introduce an aggregate weight w_t for each index tuple t satisfying

$$w_t = \sum_{t' \in c_t} w_{t'} = \sum_{\texttt{leaf tuple } t' \in \mathcal{T}_t} w_{t'} \tag{1}$$

In other words, w_t is the summation of the weights of all the leaf tuples in the subtree. In addition, if a tuple t is on page p, we denote $s_t = \sum_{t' \in p \wedge k_{t'} < k_t} w_{t'}$, i.e., s_t is the summation of all the weights of the tuples whose keys are smaller than t's on page p.

If an aggregate B-tree stores all the index tuple weights w_t exactly, we can then draw random samples by traversing down the tree from the root by making random choices of tuple t on a page p with $Pr(t) = w_t / \sum_{t \in P} w_t$. It is easy to show that each leaf tuple is sampled with the probability proportional to its weight.

However, it is not always practical to store the weights exactly. For instance, when we are inserting some leaf tuple t_0, we will need to update the weights along the search path one by one, leaving some of them with a higher weight than it should be. We will have to expose these inexact weights if we were to allow concurrent access to the tree while the update is in progress. Suppose we denote the actual weight stored in an index tuple t as $\tilde{w}_t$ (and conveniently denote the weight of a leaf tuple t as $\tilde{w}_t$ as well). Correspondingly, let $\tilde{s}_t = \sum_{t' \in p \wedge k_{t'} < k_t} \tilde{w}_{t'}$. It turns out that we can use Algorithm 1 to perform rejection sampling as long as it is consistent for sampling purposes (as defined in Definition 1).

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

Algorithm 1: Sampling an aggregate B-tree

```
Input: An aggregate B-tree $T$
1  $d \leftarrow$ random() ;                //a random number in (0, 1)
2  $p \leftarrow$ the root page of $T$ ;
3  $r \leftarrow d * \sum_{t \in p} \tilde{w}_t$ ;
4  while $p$ is an internal page do
5      if $r \geq \sum_{t \in p} \tilde{w}_t$ then
6          goto line 1 ; //sample rejection and retry
7      $t \leftarrow$ the index tuple $t$ such that $\tilde{s}_t \leq r < \tilde{s}_t + \tilde{w}_t$ ;
8      $r \leftarrow r - \tilde{s}_t$ ;
9      $p \leftarrow c_t$ ;
10 if $r \geq \sum_{t \in p} \tilde{w}_t$ then
11     goto line 1 ; //sample rejection and retry
12 $t \leftarrow$ the leaf tuple $t$ such that $\tilde{s}_t \leq r < \tilde{s}_t + \tilde{w}_t$ ;
13 return $t$;
```

In an aggregate B-tree, we must maintain consistent weights for sampling purposes. To ensure that, any insertion that increases the weights must apply the updates from root to leaf, while any deletion which decreases the weights must do so in the reverse order. This may leave some gap between the stored weight and the actual weight while the update is still in progress.

On the other hand, we also want the gap to be as small as possible to ensure high efficiency. As an example, suppose we draw samples from a table with N tuples in order to compute SUM(M) for some measure M. Let the tree fan-out be B and the rejection rate (i.e., the probability that Algorithm 1 needs to retry) be ρ. The expected number of I/Os for fetching m independent samples using an aggregate B-tree is $O(\frac{m}{1-\rho} log_B N + m)$. In contrast, a scan-based sampling algorithm always need $O(N/D)$ I/Os where D is the average number of tuples per heap page. Even if we ignore any effect from buffering, prefetching and the performance difference between random I/O and sequential I/O, we need to have a small enough ρ value to ensure aggregate B-tree based sampling has any advantage compared to scan-based sampling. In practice, $\frac{m}{1-\rho}/N$ usually needs to be less than 0.1%. Since the amount of samples m needed to achieve ε error is $O(1/\varepsilon^2)$ [6] (a constant irrelevant to N), this is achievable only if ρ is small enough.

3 DESIGN PRINCIPLES

In this section, we describe the main design principles of aggregate B-trees that deviate from conventional concurrent B-tree designs. The designs discussed in this section are generally applicable to most latch-based concurrent B-trees which modify pages in place. We will discuss practical implementation details in Section 4.

3.1 Atomic Updates to the Weights

The conventional method for updating some value on a B-tree page requires exclusive access at the granularity of page. It works for ordinary B-trees because most page updates are at or near leaf levels, and thus rarely conflict with others.

In contrast, aggregate B-trees have significantly more conflicts near the root page. Every insert or delete in the tree requires updating a weight value on each of the h pages from root to leaf along the search path. That means, every updating thread will conflict with others at least on the root page. It is also easy to show by birthday paradox that there is at least 50% chance of having an additional conflict on the second level below root between two threads at any time if there are at least $O(\sqrt{B})$ concurrent updating threads (B is the tree fan-out) – that is merely 17 threads if the tree fan-out is 300 (roughly the average fan-out of an integer-keyed B-tree in PostgreSQL with the default 8KB page size). Hence, the conventional method will result in a massive amount of waits or retries for ensuring exclusive access.

Our observation is these conflicts are actually due to the coarse granularity and the mechanism used for exclusive access rather than true conflicts. For instance, multiple threads may be updating different weight values for the insertions or deletions under different subtrees on the same root page, so they do not really conflict. In addition, weight updates are commutative, so it does not matter how they are ordered even if multiple threads are updating a single weight simultaneously. Therefore, we can use atomic Fetch-And-Add (FAA) for weight updates to ensure the correctness of weight updates. It imposes the following two requirements on the B-tree, both of which are achievable in most B-trees.

- It updates pages in place in an in-memory buffer. This is true for most B-tree designs that use conventional buffer pools to pin a page before access. After an update, we can simply mark the page as dirty to persist the data. While it marks h pages as dirty in every update as opposed to 1 in an ordinary B-tree, it usually does not impact the number of I/O as long as the buffer pool is reasonably large to buffer all the non-leaf pages. For instance, for a B-tree with 1 billion integer keys in PostgreSQL, the tree height is 4 and we only need a few megabytes of buffer space to buffer all the

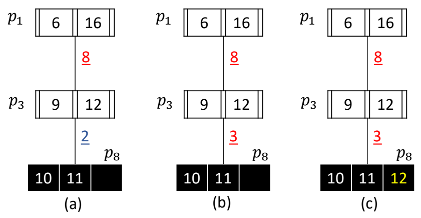


Figure 2: Updating the weights during the initial search for insertion point when there are no concurrent SMO in the inserting subtree

non-leaf pages. In the case where the internal pages do not fit in main memory (e.g., large BigTable instances [1]), this will likely to incur more I/Os per update for each read and write at the lower levels of the tree. However, this will not be a significant overhead during updates compared to non-aggregate B-trees since they both have to perform random reads on internal pages that don't fit into memory during search. Dirty pages flushing can be batched and performed in background to fully utilize the I/O bandwidth. Thus, the latency of writes can be hidden in many use cases as long as there is a sufficiently large buffer pool to fit the working set.

- The location of values must not change concurrently. To ensure that, we can acquire a read latch before updating the weight. Any thread performing SMO, which may move the value to a different address, must take a write latch instead. Since SMO is relatively infrequent, the wait resulting from a write latch is not significant. Meanwhile, all threads not performing SMO may benefit from the read latches where they can interleave their weight accesses in any order with no coordination.

3.2 Weight Updates under Concurrent Split

To consistently update the weights for an insertion, we need to increment the weights of index tuples pointing to the inserting subtrees in the root-to-leaf order. At a first glance, we can simply do that during the initial search of the insertion point before we perform the actual insertion. The updated weights along the search path are temporarily w_t greater than it should be before the actual insertion of t happens. For example, if the key of t is 12, and we have only one updating thread that is inserting t into the tree shown in Figure 1. Then any other sampling thread may only find one of the partially updated state of the tree as shown in Figure 2. All three are consistent for sampling and the final state (c) does not cause any rejection for sampling. This either requires the tree to admit one updater at a time, or use latch coupling (i.e., holding a latch on a parent page while pinning and latching its child page during the initial search), but neither is very scalable for a large number of concurrent updaters.

Inconsistent weights by unsafe concurrent splits. Many B-tree concurrency control schemes admit concurrent SMO with non-SMO operations. Notably, the B-link tree [16] and its variants allow access to a concurrently split page through horizontal links, when it has not installed any vertical pointer from the parent page to the newly created page. It is actually incorrect to use the aforementioned strategy for weight updates in this case. Figure 3 shows a scenario

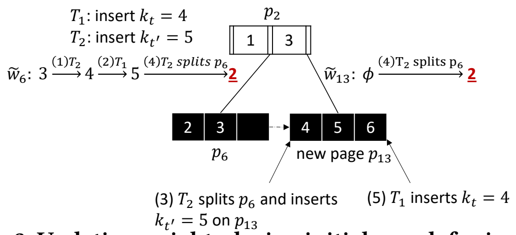
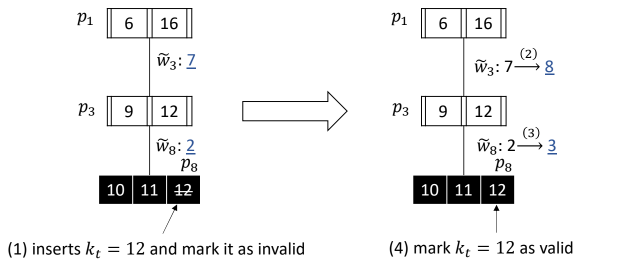

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$
p_2
1
3
$\tilde{w}_6$: $3 \xrightarrow{(1)T_2} 4 \xrightarrow{(2)T_1} 5 \xrightarrow{(4)T_2\ splits\ p_6} \underline{2}$
$\tilde{w}_{13}$: $\phi \xrightarrow{(4)T_2 \text{ splits } p_6} \underline{2}$
2
3
4
5
6
p_6
new page p_{13}
(3) T_2 splits p_6 and inserts $k_{t'} = 5$ on p_{13}
(5) T_1 inserts $k_t = 4$

Figure 3: Updating weights during initial search for insertion point may lead to undercount due to concurrent splits. ϕ denotes an uninitialized weight.

where two threads T_1 and T_2 concurrently insert two tuples t and t' with keys 4 and 5 respectively into the tree in Figure 1 and the weights end up being inconsistent in the end. More specifically, T_1 and T_2 first concurrently increment the weight of the index tuple pointing to the old p_6 (denoted as $\tilde{w}_6$) by 1. T_2 happens to obtain the write latch on the old p_6 before T_1 does. It then splits the page into two halves, p_6 and p_{13}, and inserts t' into the new page p_{13}. It also computes the total weights of p_6 and p_{13} based on their current content, which are both 2. After T_2 obtains a write latch on p_2 again, it immediately drops the latches on p_6 and p_{13}, and then installs the new links with new weight values 2 and 2. T_1 is able to reach the new page p_{13} through a horizontal link on p_6 as soon as T_2 drops the latches on them. It then inserts t on p_{13}. At this point, both inserts have been finished but the weight of the index tuple pointing to p_{13} (denoted as $\tilde{w}_{13}$) permanently undercounts by 1, making the weights no longer consistent for sampling. The crux of this issue is that T_2 is unable to determine whether there are concurrent updates to the split child pages p_6 or p_{13}. Any such concurrent updates invalidate the weight values T_2 just computed during the first half of the split, because the additional weight in the concurrent update are not included in either of these computed weights. It is incorrect regardless of where and when T_1 performs the final insertion. For example, imagine that T_1, instead of inserting $k_t = 4$, inserts $k_t = 2$ into p_6 in step 5. Then it's $\tilde{w}_6$ instead of $\tilde{w}_{13}$ that is undercounted by 1 as a result.

Safety conditions during concurrent splits. We adopt a different strategy: use a second search for a newly inserted tuple to increment the weights along the way. It assumes each key is unique, which can be achieved by simply attaching the heap record ID to the key. Figure 4 shows the update sequence when a thread T_1 inserts a tuple t with $k_t = 12$ into the tree shown in Figure 1 with no concurrent SMO in the subtree. When t is first inserted into the tree, it is marked as invalid and treated as a zero-weight tuple. Thus, an invalid tuple is invisible to any concurrent sampling threads, which is fine as it is uncommitted and should be invisible to concurrent threads anyway. For the ease of explanation, we say $\tilde{w}_t = 0$ when it is invalid, and we mark it as valid by adding w_t to $\tilde{w}_t$.

However, a concurrent SMO may still reset some recently updated weight if T_1 does not hold a latch when following a vertical or horizontal page pointer. For instance, a concurrent split by another thread T_2 on p_8 between step 3 and 4 may incorrectly reset $\tilde{w}_8$ as 2, which undoes the increment of 1 that T_1 did in step 3. It turns out that we can determine and correct the reset weights by just detecting the resetting SMOs that *happen before* we obtain the read latch on the page where we want to increment the weight, and thus we do not have to coordinate with concurrent threads for



p_1
6
16
$\tilde{w}_3$: 7
p_3
9
12
$\tilde{w}_8$: 2
p_8
10
11
~~12~~
(1) inserts $k_t = 12$ and mark it as invalid
p_1
6
16
$\tilde{w}_3$: $7 \xrightarrow{(2)} 8$
p_3
9
12
$\tilde{w}_8$: $2 \xrightarrow{(3)} 3$
p_8
10
11
12
(4) mark $k_t = 12$ as valid

Figure 4: Using a second search to perform update when there are no concurrent SMO

incomplete SMOs. In more details, a key observation is that an SMO only undoes the increment by T_1 when it is a split that happens on a page p where T_1 has not incremented the weight of any tuple (e.g., p_8 before step 4), *and* T_1 has incremented the weight of some tuple on the parent page of p (e.g., p_2 after step 3). Since we also need to increment the weight in the root-to-leaf order, it means the effect of a number of concurrent splits will always leave the tree in a state where the weights of the relevant index tuples above some level L include w_t, *and* the weights of those below L do not include w_t. And we should always find a page at level L to increment the weight on. Formalizing that, we define the safety conditions for updating the weight on a page p as below in Definition 2.

Definition 2. A read latched page p is said to be *safe* for weight update for a newly inserted tuple t, if it satisfies these conditions:

(1) The page p has a tuple that is either t itself or points to the subtree containing t.
(2) If p is an internal page, we also hold a read latch on its child page $c_{t'}$, for the t' that points to the subtree containing t.
(3) The weight of any tuple pointing to some ancestor page of t above the level of p has been incremented by w_t.
(4) The weight of any tuple that points to some ancestor page of the inserted tuple t at or below the level of p (including t itself) has either never been incremented by w_t or had a reset of the value due to SMO which undoes the previous increment.

Note that at any moment, there may only be at most one safe page for a given insertion thread T. Algorithm 2 shows how to correctly perform weight updates in the second search, if we can provide a function `EstablishSafePage(`t, p`)` that allows us to find a safe page given the inserted tuple t and a currently read latched page p that T has not updated yet. It is easy to show that when the algorithm finishes, the last safe page must be the leaf page that contains the newly inserted tuple t. By the definition of safe page, the weight of any tuple that points to some ancestor page of t above the leaf level has been incremented by w_t as well. At this point, t itself is also marked as valid in the last loop when its weight is incremented by w_t, so we have completed the weight update. The algorithm is generic to any B-tree structure, except for the function for establishing a safe page, which depends on how the B-tree design handles concurrency control during search. Usually that will involve a search along the original search path to a point where we can establish the safety conditions. We will show how this may be implemented for B-link tree in Section 4.

3.3 Weight Updates with Concurrent Merge

Pages may be merged (or deleted) during deletion if its utilization falls below a threshold (or it becomes empty) in order to balance the B-tree. There are generally three strategies: 1) lazy reorganization of the tree that is performed offline; 2) latch coupling from bottom up during recursive page merges; or 3) using two atomic steps for page merging (i.e., unlinking the page from parent and performing the actual merging) [12, 21]. The first two do not admit concurrent SMO because 1) the lazy reorganization can be done offline with database locks, and 2) latch coupling prevents concurrent splits from acquiring the latches needed. As a result, they are always safe for weight updates. However, the third strategy may introduce concurrent splits which are not safe. In more details, we first note that we need to subtract the weight of the deleted tuple from the index tuples from bottom up. Different from page splits, the weight of a merged page can be computed in one atomic step on the parent page only, which adds the weights of two merged page (stored in two adjacent index tuples) and then subtracts the weight of the deleted leaf tuple from the sum. As a result, it is impossible to accidentally undo a decremented weight due to a concurrent merge. However, it is still possible that we have a concurrent split after a page merge but before we manage to decrement the weight on the parent level, which may prematurely decrement the weight of an index tuple before the deletion has a chance to do so, resulting in an undercounted weight. Since it is the concurrent split that may cause undercounting, we can use the same mechanism as in page split to establish the safety conditions for weight update before decrementing the weights (with a slightly different definition of safety where any level below l is updated while any level at or above l is not).

Algorithm 2: Incrementing weights with safe page

```
Input: t: a copy of the newly inserted tuple
1  $p \leftarrow$ root of the the tree ;
2  Latch $p$ ;
3  while true do
4      $p \leftarrow$ EstablishSafePage($t$, $p$) ;
5      $t' \leftarrow$ the ancestor tuple of $t$ on $p$ (maybe $t$ itself);
6      fetch-and-add($\tilde{w}_{t'}, w_t$) ;
7      if $p$ is a leaf page then
8          Unlatch $p$ and break;
9      else
10         Unlatch the $p$ and set $p$ to $c_{t'}$;
```

4 AB-TREE DESIGN

In this section, we describe the details of AB-tree, an implementation following the design principles in Section 3, based on the B-link tree variant in PostgreSQL. In addition, we discuss how to reduce sample rejections in order to improve sampling efficiency under MVCC in general and in PostgreSQL at the end of this section. AB-tree extends the data layout with additional fields to help correctly maintain the weights and perform sampling. As in an ordinary B-link tree, the pages in each level are linked as a linked list (Figure 5). Each page p, except the rightmost page on each level, has a right sibling page p_r. In addition, each page stores a high key k_p (the additional key in the gray boxes), which is an upper bound of the keys of all leaf tuples under p and a strict lower bound of the keys in its right sibling p_r. The high key of the rightmost pages on each level is always $+\infty$.

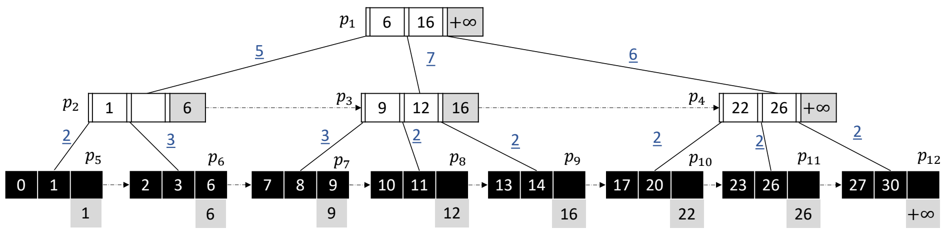


Figure 5: An aggregate B-link tree (AB-tree) with height 3 and a fan-out of 3. The dashed arrows represent the next page links in per-level linked lists. High keys of the pages are shown in gray boxes.

Note that key search, insertion/deletion of a leaf tuple, SMO, as well as how latches are used is the same as an ordinary B-link tree – we only add an additional sampling routine and an additional weight maintenance routine for updates. The following are a few key facts of B-link tree that we heavily rely on in AB-tree implementation: 1) B-link tree always does latch coupling from bottom up and from right to left (which makes SMO easy). That means, we cannot hold any latch when moving from a parent page to a child page, or from a page to its right sibling for deadlock avoidance; 2) This variant of B-link tree allows page split but not merge. A page is only deleted from the tree when it is completely empty. Hence, we only need to consider page split for SMO; 3) The key range of a tuple t may only move to the right of its original position during a page split. As a result, we can always find a search key k in the key range of some tuple t on a page p or to its right, if p was supposed to contain k in its key range before we drop the latch on its parent.

In the remainder of this section, we describe the data layout in AB-tree, as well as the implementation details for sampling and update in AB-tree. While the implementation discussed in this section is tailored towards B-link tree, it is also possible to adapt them to apply to other B-trees not having all of these particulars. In particular, one can usually replace the moving-right operation in B-link tree with a new tree search from the root in other B-trees. The Recompute ID and Structural Modification ID introduced later can also be used for detecting concurrent splits and SMO in other B-trees. If the B-tree allows merge, then one needs to establish the safety conditions for weight updates using the Recompute ID and Structural Modification ID in the deletion as well.

4.1 Data Layout

An index tuple t is a quadruple $t = (k_t, c_t, \tilde{w}_t, \texttt{RID}_t)$, where k_t is its key, c_t is a pointer to its child page, and $\tilde{w}_t$ is the stored weight. In addition, we store a 16-bit counter $\texttt{RID}_t$, or the *Recompute ID*, in t, which is incremented whenever $\tilde{w}_t$ is reset by some SMO on c_t that recomputes a new value based on c_t. Note that we normally only increment or decrement $\tilde{w}_t$ for most operations and the recompute only happens for a split. Therefore, changes in $\texttt{RID}_t$ may be used for detecting a concurrent split on c_t, which is the only operation that may make the weight update incorrect.

A leaf tuple t is also a quadruple $t = (k_t, ptr, w_t, \texttt{xmin}_t)$, where k_t is its key and ptr is a pointer to its heap tuple. We do not have to store the weight $w_t = f(t)$ in t unless the weight function f is expensive to evaluate (e.g., access to the heap tuple). The leaf tuple also stores a transaction ID, $\texttt{xmin}_t$, which is set to the top transaction ID that inserts this index tuple once insertion of t is completed. It serves two purposes: as the creation timestamp described in Section 4.4 and as an indicator for whether the tuple is valid. To mark t as invalid, we store an invalid `xmin`, and then t is treated as a zero-weight leaf tuple regardless of what w_t is.

In the header of an index page p, we store another 16-bit counter $\texttt{SID}_p$, or *Structural modification ID*, which is incremented every time there is an insertion or deletion on p. Note that a split on p also increments $\texttt{SID}_p$ as there are deletions. It exists mainly for fast path of common cases in weight updates rather than correctness: if we find $\texttt{SID}_p$ does not change after we unlatch and latch p again, we know there is not any concurrent SMO of itself and/or any of the immediate child pages. As we will show later, this can save additional accesses to the parent/ancestor pages in most cases when we try to establish safe pages for weight updates and can also save a search on page looking for the correct index tuple to update.

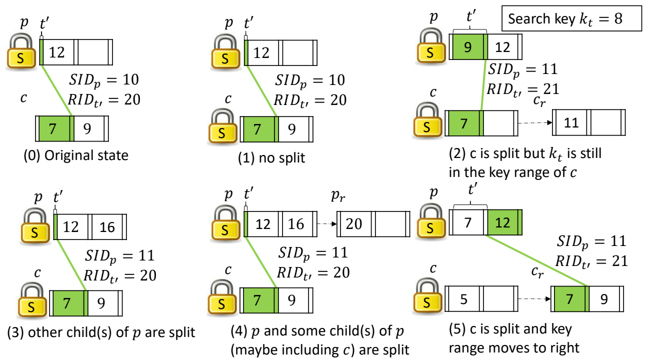
Note that neither `RID` or `SID` needs to survive a crash because they are used for detecting concurrent SMO among running transactions. Hence, they are not write-ahead logged. We do mark the buffer as dirty if we update them though, as they need to survive through page swaps without system crash. They can wrap around for being only 16-bit long, but there is unlikely any false negative in the detected concurrent SMO during a short period of time.

4.2 Sampling in AB-tree

Sampling in AB-tree is similar to Algorithm 1 but requires some slight modification to be correct. The search in B-link tree always drops the latch on parent page before we latch its children. However, a concurrent split may happen on the child page before we successfully latch it. As a result, we cannot immediately reject the sample even if we find that the total weight of the child page is smaller than the one stored in the index tuple as some of the tuples with non-zero sampling probabilities that are supposed to be on the child page may have been moved to the right. To deal with that, we save the expected high key (which can be found as the next index key or the high key on the parent page), before we move to the child page. Then we need to move right until we find a page with the matching high key. We only reject the sample if the total weights of the pages we visited in the current level is smaller than the weight stored in the index tuple in the previous level.

4.3 Updates in AB-tree

Deletion in AB-tree is actually easier than insertion, as we can use latch coupling when we decrement the weights from leaf to root. This prevents any concurrent SMO from undoing the decrements. Note that deletion in PostgreSQL only happen in batches during vacuum of dead versions. Thus, we can decrement the weights in one pass from leaf to root for a batch of tuples deleted from the same leaf page, making it more efficient.



p
t'
12
$SID_p = 10$
$RID_{t'} = 20$
c
7
9
(0) Original state
(1) no split
Search key $k_t = 8$
$SID_p = 11$
$RID_{t'} = 21$
c_r
11
(2) c is split but k_t is still in the key range of c
16
(3) other child(s) of p are split
p_r
20
(4) p and some child(s) of p (maybe including c) are split
5
(5) c is split and key range moves to right
S

Figure 6: Possible state of p and its child page c when we latch c and p again after dropping the latch on p

For the remainder of this subsection, we only consider the harder insertion case. As discussed in Section 3.2, we perform an insertion in 2 phases: the first phase inserts the leaf tuple; the second phase performs a search from the root to leaf for the newly inserted tuple in order to update the weights along the path. The key is to design the algorithm for establishing safe pages shown as Algorithm 3. In addition to a copy of the inserted tuple t, the next potential safe page p (which is either the root or the child page of the last safe page whose key range contains k_t), it is also provided with a pointer to an initially empty `stack` (reset with each invocation of Algorithm 2). This stack is used to save information about the past safe pages to help identify the next safe page. More specifically, we push the page number p, $\texttt{SID}_p$ and $\texttt{RID}_{t'}$ for the tuple $t' \in p$ s.t. $t \in \mathcal{T}_{t'}$ onto the stack whenever we return a safe page p.

Different states after page split. The algorithm first locates a tuple t' whose key range matches the key of the inserted tuple t (i.e., $t \in \mathcal{T}_{t'}$). Then it tries to move to its child page $c = c_t$. We must drop the latch on p before latching the child page c (line 9) due to the latch coupling order of B-link tree, so it is possible that there are concurrent split(s) on p, c, and/or some other immediate child page(s) of p. Figure 6 shows all the 5 possible scenarios that could result from the original state after line 10. Case 1 is the most common case where no split happens on p or its child pages, and thus p is safe for return. It is the only case where $\texttt{SID}_p$ does not change, so we can quickly determine that without accessing additional pages on the stack (line 11). In the remaining four cases, p is not safe in two of them: case 4 for the split of p undoes the increment of w_t in p's parent (or in the case of root split, the new root does not count w_t either), violating the safety condition (3), and case 5 for the split of c moves the subtree containing t to its unlatched right sibling, violating safety condition (2). They may happen at the same time: a split on c that moves the subtree containing t to the right may also lead to a split on p. If case 4 happens, the safe page must be above p. Otherwise, the safe page could still be at the same level as p if we can latch the correct child page later. In the algorithm, we can determine case 5 easily using the condition on line 13: either we cannot find a $t' \in p$ such that $t \in \mathcal{T}$, or t' no longer points to c (line 13). Then we need to determine whether case 4 happens in order to decide how to reestablish the safety condition, which requires checking the `SID` and `RID` of the parent of p.

Identifying concurrent page split. The definitive way to determine whether p is split is to find the index tuple t' that points to p

Algorithm 3: Establish a safe page for weight update

```
Input: $t$: a copy of the newly inserted tuple
  $p$: the latched root or the latched child page of last safe page
  stack: a stack of triplets $(p', \texttt{SID}_{p'}, \texttt{RID}_{t'})$ for previously
  returned safe pages $p'$ and $t' \in p'$ s.t. $t \in \mathcal{T}_t'$
1  def PushStack($t, p$, stack)
2      find $t' \in p$ s.t. $t \in \mathcal{T}_{t'}$ ;
3      push $(p, \texttt{SID}_p, \texttt{RID}_{t'})$ onto stack ;
4  def EstablishSafePage($t, p$, stack)
5      if $p$ is a leaf page then
6          return $p$
7      PushStack($t, p$, stack) ;
8      $c \leftarrow c_{t'}$ for the $t' \in p$ s.t. $t \in \mathcal{T}_{t'}$;
9      Unlatch $p$;
10     Latch $c$; Latch $p$ again ;
11     if $\texttt{SID}_p$ = stack.SID then
12         return $p$ ;
13     if $\nexists t' \in p$ s.t. $t \in \mathcal{T}_{t'} \vee c_{t'} \neq c$ then
14         Unlatch $c$; $c \leftarrow null$ ;
15     pop(stack) ;
16     while stack is non-empty do
17         $p' \leftarrow$ stack.p and latch $p'$ ;
18         if stack.SID = $\texttt{SID}_{p'}$ then
19             nosplit $\leftarrow$ true ;
20         else
21             while $\nexists t' \in p'$ s.t. $c_{t'} = p$ do
22                 $r \leftarrow$ the right sibling of $p''$ ;
23                 Unlatch $p'$; latch $r$; $p' \leftarrow r$ ;
24             find the $t' \in p'$ s.t. $c_{t'} = p$ ;
25             if stack.RID = $\texttt{RID}_{t'}$ then
26                 stack.p $\leftarrow p'$; stack.SID $\leftarrow \texttt{SID}_{p'}$ ;
27                 nosplit $\leftarrow$ true ;
28             else
29                 nosplit $\leftarrow$ false ;
30         if nosplit then
31             Unlatch $p'$ ;
32             if $c \neq null$ then
33                 PushStack($t, p$, stack);
34                 return $p$ ;
35             else
36                 return EstablishSafePage($t, p$, stack) ;
37         Unlatch $c$; $c \leftarrow p$; $p \leftarrow p'$; pop(stack) ;
38     if $p$ is still a root then
39         return $p$ ;
40     Unlatch $p$; Unlatch $c$ ;
41     $p \leftarrow$ root of the tree and latch $p$;
42     return EstablishSafePage($t, p$, stack) ;
```

and check whether $\texttt{RID}_{t'}$ has changed since last time we established safety on the page where t' was. To do so, we pop p from the stack and latch the page p' last returned as the safe page. Since p' may be concurrently split as well, p' may not be the parent page of p at this time and thus we may have to move to its right to find the index tuple t' that points to p. To save the time for searching for t', we first check whether $SID_{p'}$ has changed (line 18). If not, there is no SMO at all on any immediate child page of p', which includes p, and we can avoid the searches for t' and the checks on the RID. Otherwise, we may move to the right of p' until it finds t' (line 21-23). After these checks, if we find that p was not concurrently split, we can establish that either p is a safe page if it's not case 5 (line 33-34), or we have the same precondition as the beginning of `EstablishSafePage()` if it's case 5 (line 36). On the other hand, if we find that p was concurrently split, the safe page must be above p. Luckily, we essentially have the same precondition as the beginning as if we have excluded case 5. At that time, we have latched p's child page c whose key range contains t; we know that t' pointing to c does not have its weight incremented yet; and we only need to check whether p is split to decide whether it is safe or not. Replacing p with p' and c with p, the preconditions listed above are exactly the same. Hence we can use the same checks above iteratively until we find a safe page. A corner case (line 38-42) is that we have an empty stack when we need to check whether p is split or not. If p is still a root, then it is not split since PostgreSQL never deletes a root. Otherwise, there was a root split on p which definitely did not count the w_t in the new root and we need to restart from the root.

It is easy to show that successive calls to Algorithm 3 always return the safe page for weight update, until it returns the leaf page containing t as the final one. It may have to go back to a page in a higher level when some SMO undoes the weight update. It, however, should be quite rare because SMOs are mostly in the lower levels near leaf and thus rarely conflict with concurrent weight updates.

4.4 Reducing Sample Rejections due to MVCC

DBMS often uses Multi-Version Concurrency Control (MVCC) to increase concurrency by allowing transactions to run on different snapshots. This is especially true for read-only analytical queries which tend to access a large amount of data as it can avoid unnecessary blocking or aborts due to read-write conflicts. Under MVCC, insertion creates a new version of the tuple tagged with a creation timestamp. Deletion only marks an old version with a timestamp indicating when it is deleted. Garbage collection of an old version happens after it is no longer visible to any concurrent transactions. A B-tree index is usually agnostic to MVCC. In other words, every new version gets inserted into the tree without removing the old one. Any tuple returned by the tree requires a separate fetch from the heap to check whether it is visible to the running transaction. For sampling, an invisible tuple is the same as a sample rejected during the sampling process, which will cause a retry to draw a new sample from the root. It effectively decreases the sample acceptance rate by an additional multiplicative factor $1 - \tau$, where τ is the percentage of the weights of invisible tuples that may be sampled by the tree. For the same reason mentioned in Section 2.2, a higher τ value will lower the break-even threshold for aggregate B-tree based sampling to be faster than scan-based sampling. Therefore, we want to reduce τ by avoiding returning invisible versions.

There are two sources of invisible versions: 1) dead versions that are invisible to any live thread and 2) live versions that are newer than the snapshot the sampling process is running on. Fortunately, the dead versions may be frequently cleaned by the DBMS if its quantity exceeds a certain threshold, so that won't be a major source of the increased rejection rate. The live versions, however,

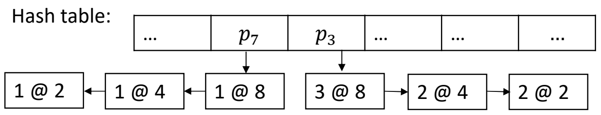


Figure 7: Multi-version weight store

may not be cleaned at all because they are newer versions that replace those in the older snapshot the sampling is running on. This cause a snowball effect on the rejection rate of a sampling thread: the longer it runs, the more newer invisible live versions there will be, the higher rejection rate becomes.

In order to avoid returning invisible live versions in a sampling thread, we introduce a light-weight multi-version weight store that allows computation of a tighter upper bound of the weight under an older snapshot. Here we make two assumptions about the MVCC model: (1) it is able to obtain the creation timestamp for an insertion at any time with very low cost, even if the transaction has not committed; (2) it is able to assert a versioned tuple being invisible to a snapshot solely based on its creation timestamp (but not necessarily able to assert visibility). These assumptions are easy to satisfy. For instance, a very common MVCC model is that a transaction may run at a start timestamp, but commit at a later commit timestamp. All versions committed by the transaction are treated as created at the commit timestamp. In this case, a versioned tuple is invisible to any transactions started before its commit timestamp, and the start timestamp can be used in place of the commit timestamp as a lower bound for asserting invisibility. In the more complicated MVCC model of PostgreSQL, a transaction commits at a timestamp obtained upon its first write, which is available at the time we insert the tuple into the tree, while a snapshot (logically) becomes a list of timestamps of concurrent transactions. Yet one can still assert a tuple is invisible to a snapshot if its creation timestamp is in the list of concurrent transactions.

With these assumptions in mind, we now describe how the multi-version weight store works (Figure 7). For each index tuple t, we maintain an in-memory linked list (called the version chain of t) of delta weight and creation timestamp pairs in decreasing timestamp order. These version chains are stored in a central in-memory hash table. Both the hash table and the linked lists are implemented using the lock-free linked list in [10] to avoid locking overhead over these heavily contended data structures. We use the page number of the child page c_t as the key rather than the physical record ID of t because it should not change across SMO. When a thread increments $\tilde{w}_t$ for an insertion, it simply prepends a pair of the creation timestamp and delta weight it added to $\tilde{w}_t$. In the case of a page split on c_t, the original version chain of t is replaced by a new version chain, which may be constructed from all version chains of the index tuples on c_t if c_t is an index page, or from the creation timestamp of all the leaf tuples on c_t if it is a leaf page.

When a sampling thread reads $\tilde{w}_t$, it scans the version chain and subtracts all the delta weights from $\tilde{w}_t$ with creation timestamps that are invisible to its snapshot. As a concrete example, suppose we have a sampling thread running on a snapshot with a start timestamp of 5 in the simple MVCC model. When it reads the weight of t with $c_t = p_7$, it will subtract the 1 at the creation timestamp 8 from the current value $\tilde{w}_7$ stored on the page. Similarly, if it runs on a snapshot with a start timestamp of 3, it will subtract both 1's at the creation timestamps 4 and 8. Actually, we do not need to scan

```
SELECT COUNT(*) FROM A TABLESAMPLE SWR(?);
SELECT COUNT(*) FROM A TABLESAMPLE BERNOULLI(?);
INSERT INTO A VALUES (?, ?);
DELETE FROM A WHERE Y = ?;
```

Figure 8: The prepared statements used in the experiments

the entire chain. Rather it may be able to stop at the start timestamp (for simple MVCC model) or the smallest concurrent transaction timestamp (for PostgreSQL MVCC model) in the snapshot, which indicates we cannot assert invisibility over any timestamps in the rest of the chain. We design the version chains in such a way that achieves a good trade-off between update/storage cost and how close the reconstructed weight is to the actual weight. We do not maintain the exact commit timestamp and deletion timestamp in the chain, which not only occupy more space, but are also very expensive to compute as it requires a transaction to retroactively update all the chain nodes upon commit and/or abort. On the other hand, the gap between the reconstructed weight and the actual weight is limited to a small fraction because these are induced by either false negatives from concurrent transactions of the snapshot (which are no more than a small constant), or dead versions from deletion and aborts which are automatically cleaned by the system when its quantity exceeds some threshold.

Another note is that the multi-version weight store never needs to survive system crash since only concurrent sampling threads need it. The version chains may also grow indefinitely if we do not remove version with creation timestamps not concurrent to any live and future snapshots. Therefore, we have a dedicated epoch-based garbage collector that periodically computes a safe upper bound of the creation timestamps that are no longer needed, and recycle all the nodes with creation timestamps lower than the bound.

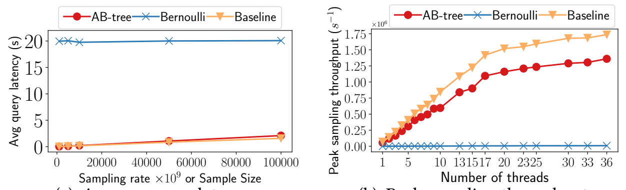
5 EXPERIMENTS

In this section, we provide an empirical study on the performance of AB-tree against its baseline (the exclusively latched aggregate B-tree described in Section 2) in terms of their concurrent sampling and update performance.

5.1 Experimental Setup

We implement AB-tree based on the B-link tree implementation in PostgreSQL 13.1 [8]. Each leaf tuple has a constant unit weight of 1 so that sampling in the AB-tree returns independent uniform samples. In addition, we implement a new `TABLESAMPLE` operator `swr`(m) returning m independent samples using the AB-tree with replacement. For each sample, it generates an independent uniform random number in $(0, 1)$ and uses Algorithm 1 to draw a sample. If that sample fails the visibility test, it automatically retries.

System Configuration. We build our customized PostgreSQL using GCC 9.3 with -O3 flag on Ubuntu 20.04. The system is configured with 2 Intel Xeon E5-2697 v4 CPU (each with 18 cores and hyperthreading enabled) and 256 GB RAM in total. During tests, we pin all the PostgreSQL processes and its shared memory to one of the NUMA node to reduce the impact of NUMA. The PostgreSQL data files are stored on a Samsung 970 EVO 1TB SSD and the Write-Ahead Logs are separately stored on a SanDisk SDSSDH3 2TB SSD. The buffer pool is set to 32GB unless otherwise specified. We set the maximum WAL segment size to 48GB to prevent checkpointer's disruption. We also disable the full page write feature to avoid the occasional forced full page image log entry which may add



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

Figure 9: Sampling with varying number of threads

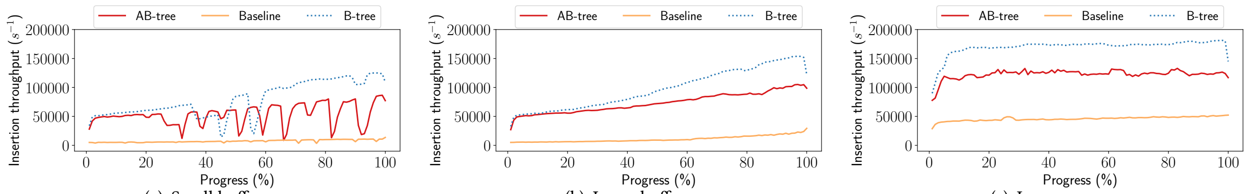
significant irrelevant I/O traffic. The multi-version weight store is configured with 2^{20} buckets and the GC thread runs every 10 seconds. The hash table entries and nodes in the chains of weight update history are 24 bytes in size and are both allocated from a fixed-size object pools backed by POSIX shared memory initialized to 2 GB. It may map additional shared memory on demand in theory but we never experienced so in the experiments. All the queries are submitted through JDBC using prepared statements (shown in Figure 8) to the PostgreSQL instance running at local host. The inserting client commits after every 10K insertions while the sampling client automatically commits for every SELECT statement.

Dataset. Most of the experiments run on an AB-tree index on the y column of a table $A(x : \texttt{INT4}, y : \texttt{INT4})$, with 64-bit integer weights. We initially load the table A with 1 billion random tuples unless stated otherwise. All the experiments are run on a fresh copy of the database. Note that we deliberately use a narrow synthetic table in order to minimize the heap access overhead when measuring the index performance. The heap file of A is about 33.8 GB, a loaded AB-tree is 28.9GB. The original B-link tree and the baseline aggregate B-tree (described in Section 2) built on the same table are about 20.4 GB. The space for storing weights in the index tuples is negligible as the number of them is < 0.3% compared to the number of leaf tuples. The increase in AB-tree's space usage is due to the additional 4-byte $\texttt{xmin}_t$ stored in the leaf tuples, which has to be aligned to the even larger 8-byte boundaries due to PostgreSQL restrictions. All these indexes have a fan-out around 300 and 4 levels in height. In addition, we run a more realistic mixed workload over the lineitem table of TPC-H with scale factor 100 in order to demonstrate the actual gain one may expect from AB-tree.

5.2 Sampling Performance

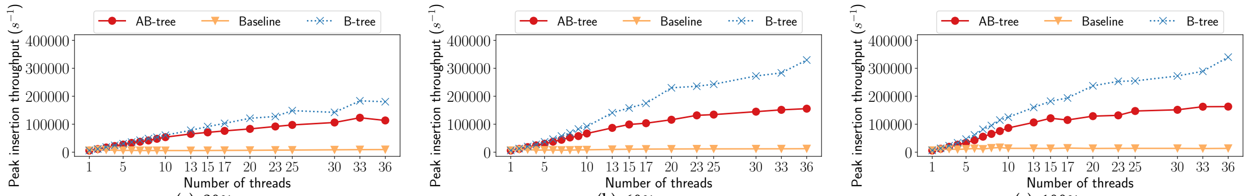
We first evaluate the sampling performance of AB-tree against the baseline, and the SQL Bernoulli sampling. Here we do not flush the OS I/O cache after copying the data. This mimics a typical scenario where one loads the data and then immediately issues many queries.

We first compare the average query latency with different sample sizes using a single thread. For AB-tree and baseline, we continuously issue the same query for about 30 seconds and take the average query latency. For the Bernoulli sampling, we only run it for 10 times because of its significantly higher cost. As shown in Figure 9(a), AB-tree has a query latency that ranges from 0.02 second (1000 samples per query) to 2.1 seconds (100,000 samples per query) because it sustains a sampling throughput of 48K samples per second. Note that the baseline does not have the overhead of accessing the multi-version weight store. Hence, it has a slightly higher throughput (about 60K samples/second) and a slightly lower query latency from 0.016 second to 1.6 seconds. In contrast, the SQL Bernoulli sampling has a running time of around 20 seconds regardless of the sampling rate, which is 2 - 4 orders magnitude slower than aggregate B-tree based sampling. Since AB-tree's query



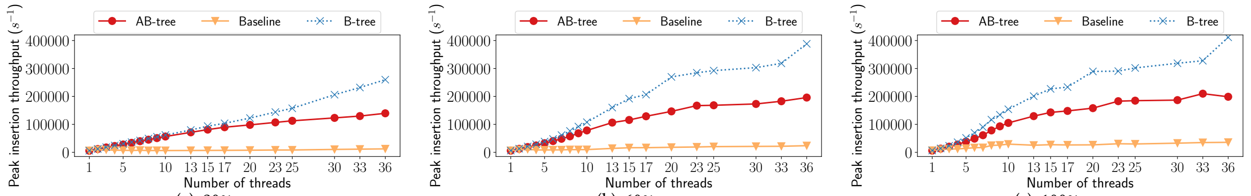
(a) Small buffer (b) Large buffer (c) In-memory

Figure 10: Insertion throughput over time with 10 threads



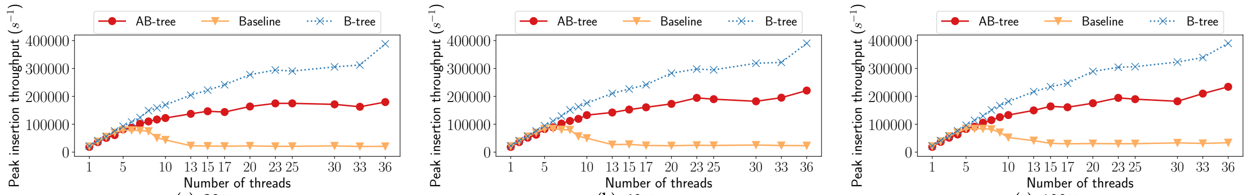
(a) 20% (b) 60% (c) 100%

Figure 11: Insertion throughput vs number of threads with small buffer



(a) 20% (b) 60% (c) 100%

Figure 12: Insertion throughput vs number of threads with large buffer



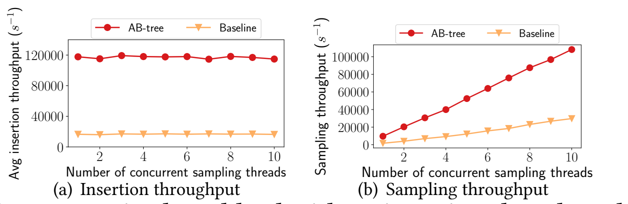
(a) 20% (b) 60% (c) 100%

Figure 13: Insertion throughput vs number of threads with all pages buffered in memory

latency increase linearly to the sample size, it can be predicted that it has a shorter sampling query latency than the Bernoulli sampling as long as the sample size is smaller than 10^6 (0.1% of the table). Note that in a more realistic wide schema, the gap between Bernoulli sampling and aggregate B-tree-based sampling will be even bigger due to the larger heap files. Figure 9(b) measures how the sampling throughput scales with more sampling threads. Here, we run up to 36 concurrent threads for 30 seconds with AB-tree and the baseline, and 10 times for Bernoulli sampling. In particular, we measure the peak sampling rate by aggregating the number of samples produced by all threads and find the highest moving average of the number of samples produced in a window size of 10 queries asking for 10,000 samples (which is roughly 2 seconds for AB-tree, 1.75 seconds for the baseline and 198 seconds for the Bernoulli sampling). We take the moving average since we record the running time for every 10,000 samples but report the total number samples produced every 1/4 second. Without the moving average, a query longer than 1/4 second would appear to finish within a single 1/4 second window. Consistent with the previous experiment, aggregate B-tree based sampling can achieve at least 2-3 orders of magnitude higher throughput than Bernoulli sampling. Even with a single thread, both AB-tree and the baseline can achieve a sampling rate around 50K-60K, while the Bernoulli sampling can only achieve 500 samples/second. Comparing AB-tree with the baseline, accessing the multi-version weight store does poses about 30% overhead to the sampling throughput in AB-tree but, as we will show later, this trade-off enables AB-tree to maintain a high throughput for both sampling and update concurrently. Moreover, we can always disable the multi-version weight store to eliminate the overhead if there are few updates. Another note is that the scalability of AB-tree and baselines gets throttled above 18 threads. The reason for that is the CPU we used only has 18 physical cores and hyperthreading cannot fully pipeline a memory-access-heavy workload. As this is an intrinsic property of sampling and index update workloads, all the scalability tests in this section inevitably suffer from the same hyperthreading issue, which we believe is a common issue to many concurrent index designs and deserves further investigation, especially for many-core systems.

5.3 Concurrent Insertion Performance

Next, we compare concurrent insertion performance of AB-tree and the baseline by inserting up to 500,000 random tuples per thread into the database. We also include the original B-link tree in PostgreSQL into the comparison, as its performance is a **strict upper bound** of any aggregate B-tree. As insertion performance is highly dependent on the a variety of factors, including the size of buffer pool and system I/O cache, we test three common scenarios: (1) (**small buffer**) loading a cold database that has a very small



(a) Insertion throughput (b) Sampling throughput

Figure 14: Mixed workload with 10 insertion threads and varying number of sampling threads

buffer pool; (2) (**large buffer**) loading a cold system whose buffer pool is not enough for caching all pages; (3) (**in-memory**) loading a database fully buffered in the memory. In scenario 1, the system is configured with a 128 MB buffer pool and we flush the system I/O cache beforehand as well. In scenario 2, the system is configured with a 32 GB buffer pool, which is enough to buffer all internal pages over time but not all leaf pages and heap pages. Scenario 3 is simulated by generating the same random tuples as scenario 2, and insert them into a loaded scenario 2 database. Hence, the majority of pages we access in scenario 3 are already in the buffer pool.

Figure 10 shows how the insertion throughput changes over the entire course of execution with 10 threads. Because the running time is different with various indexes, the x-axis is the percentage of time elapsed and y-axis approximates its instant throughput by averaging the preceding 2-second window. When the buffer pool is small, there are frequent drops in throughput due to I/Os for swapping out dirty pages. In all three scenarios, AB-tree outperforms the baseline by 4-11x in insertion throughput and the gap becomes larger as more pages are buffered. B-link tree's performance is an upper bound to achieve, and we can see that AB-tree's overhead of updating weights is only about $\frac{1}{3}$.

We also show scalability against the number of threads for AB-tree, the baseline and theoretical upper bound provided by B-link tree up to 36 threads. Since the insertion throughput increases over time as more data pages are buffered (as shown in Figure 10), we compare the peak throughput at different progress in time. Figures 11, 12, 13 show the peak throughput at 20%, 60% and 100% progress for the small buffer, larger buffer and in memory scenarios. In all cases, AB-tree scales much better than the baseline in terms of the insertion throughput. When data are not buffered in memory and a lot of disk I/Os are involved, the exclusively latched path significantly restricts concurrency for the baseline, which peaks at around 13K insertions per second and does not scale beyond a few threads. In contrast, AB-tree is able to scale similarly to B-link tree, albeit with a 1/3 overhead. It can reach a throughput of about 163K insertions per second. When there are fewer I/Os in the large buffer and in memory cases, the performance gap is a bit smaller. Nevertheless, AB-tree consistently outperforms the baseline by at least 3-7 times in all experiments with more than 10 threads, regardless of the buffer pool size.

5.4 Read-Write Workload

Next, we study how AB-tree performs on mixed read-write workloads. The first case is a mixture of concurrent sampling queries and insertions. This is a typical scenario where users try to do analysis in real time while there are ongoing data ingestion. In this set of experiments, we set the buffer pool size to 32GB and warm it up by running 60 seconds of sampling queries beforehand. We run 10 concurrent insertion threads along with up to 10 sampling threads.

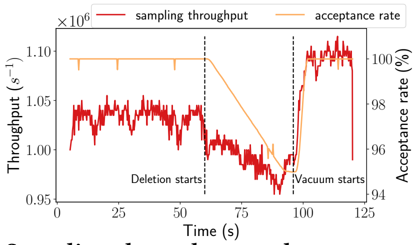


Figure 15: Sampling throughput and acceptance rate trend with deletion and vacuum

The sampling threads continuously issue queries asking for 10000 samples. Since we have found that the instant throughput of insertion or sampling are pretty stable in a database with warmed buffer pool, we measure the average insertion and sampling throughput with different numbers of sampling threads.

As shown in Figures 14(a) and 14(b), insertion throughputs of both AB-tree and baseline are quite stable as the number of query threads grows. This is not surprising because the sampling process only holds one latch at a time for a relatively short period of time. The insertion throughput of AB-tree is roughly the same as that of 10 threads with large buffer as we find in Figure 12(c), but the baseline's insertion throughput decreases by 40% compared to its counterpart in Figure 12(c). In addition, it is unsurprising that the sampling throughput is reduced for both due to exclusive latches held during insertion. However, AB-tree's sampling throughput under active insertion only reduces to $\frac{1}{5} \sim \frac{1}{6}$, while that reduces to $\frac{1}{30} \sim \frac{1}{40}$ for the baseline, clearly showing AB-tree's advantage.

We also evaluate the impact of deletion and vacuum on concurrent sampling. It simulates the scenario when a background data cleaning thread identifies and removes stale data from the database while one concurrently analyzes the data with a number of sampling queries. Since the effect of dead tuples on sampling efficiency is not obvious if we only delete a relatively small number of tuples, we loaded another database with 10 million random tuples instead of 1 billion. We disable the autovacuum of PostgreSQL and run 10 concurrent sampling threads for 120 seconds. We measure the sampling throughput as well as the overall sample acceptance rate. At 60s, we launch a thread that deletes about 5% of all tuples. It finishes at around 91s. Then at 96s, we launch another thread to run vacuum against the database, which runs for 3 seconds. Figure 15 shows the results. When the deletion starts, the acceptance rate gradually drops to about 95% as there are more dead tuples in the table. The sampling throughput suffers an immediate drop because the sampling threads have to switch to index scan instead of index-only scan due to visibility checks. Then, its drop reaches the same percentage when the delete finishes. Once the vacuum starts, the acceptance rate rises until it reaches 100%. The sampling throughput also recovers along with the acceptance rate and it is actually slightly higher than it is before the vacuum. That is attributed to the fact that there are fewer tuples remaining in the database.

5.5 Skewed Workload on TPC-H

Finally, we evaluate AB-tree in a more realistic setting using the TPC-H dataset with scale factor 100. We load 3 databases with the lineitem table, which has about 600 million tuples, and build the AB-tree, the baseline aggregate B-tree, and the regular B-link tree respectively over `l_shipdate`. We use the AB-tree and the baseline

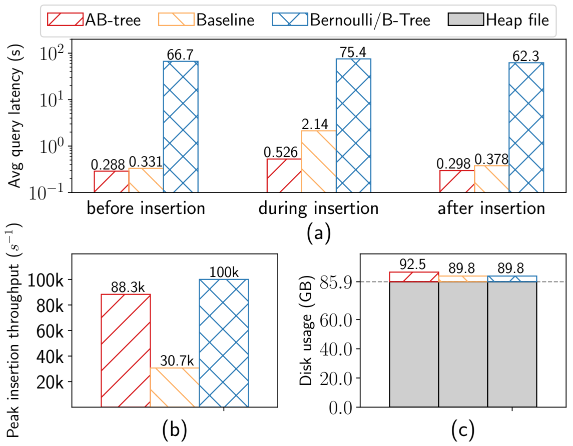


Figure 16: Mixed workload on TPC-H lineitem (SF = 100)

aggregate B-tree to draw random samples in the first two databases. In the third, we use the SQL Bernoulli sampling with an appropriate sample rate to draw roughly the same amount of samples. We simulate a more realistic scenario for real-time data analytics with data and query range skews. More specifically, we set up 20 query threads representing the data analysts who are analyzing the total revenue of sales for items shipped within a random 1-year period using 10,000 random samples. We skew the query ranges by randomly generating 90% of the ranges within the last 2 years in the data, with the remaining 10% uniformly distributed in all possible date ranges. In the meantime, 10 concurrent insertion threads insert new tuples into lineitem. 99% of the new tuples have ship dates within the last 183 days from the last day in the dataset, with the remaining 1% uniformly distributed in the entire 7-year range.

Figure 16 shows average query latency before/during/after the insertions, peak insertion throughput, and disk space usage. Note that we include the cost of maintaining B-tree in the evaluation for Bernoulli sampling even if the PostgreSQL does not choose index scan for large ranges. The rationale is that one may still want to keep the B-tree in case of the need to run zoom-in queries on smaller ranges. We find that AB-tree achieves the best trade-offs with this workload. The gap between AB-tree and the baseline in terms of insertion throughput is much bigger compared to the previous scalability tests, while its gap to the regular B-link tree is becoming smaller. AB-tree is also able to consistently maintain a low query latency with random sampling. Notably, its query latency is about 2 orders of magnitude lower than Bernoulli sampling and 4x lower than the baseline during insertions. This experiment demonstrates that AB-tree can achieve a very good end-to-end performance gain and a moderate maintenance overhead in a realistic workload. The space overhead of AB-tree relative to the heap file size, is quite tolerable (about 7.6%), which is merely 3.1% more than the baseline and B-tree relative to table size. We do not include a figure for how the relative index sizes change with different heap file sizes, because all three ratios will remain the same. Moreover, the wider the table is (as most real datasets are), the space overhead is more negligible.

6 RELATED WORKS AND EXTENSIONS

There are a plethora of related works on designing concurrent B-trees in the literature. Lehman et al. first designed the B-link tree [16] that supports high concurrency for updates and queries. Many B-link variants tree are proposed to improve the concurrency with different locking protocols, reduce the tree imbalance cause by concurrent delete, recover from crashes [12, 15, 16, 20, 26, 27].

There have also been efforts in building latch-free B-trees to support higher level of concurrency on modern storage devices. Examples include [18, 29] for Solid State Drive and [4] for Non-Volatile Memory. These works utilize atomic operations like compare-and-swap to implement SMOs. It is unclear whether we can build aggregate B-trees on top of these latch-free B-trees as we need to fixate the location of stored weights when incrementing or decrementing the weights. It would be a possible direction to explore maintaining the stored weights outside the tree nodes or pages. In principle, any index structure can be augmented with aggregate weights to assist sampling. There are also plenty of works discussing how to build other concurrent index structures efficiently using hash tables, skip lists, trie trees [7, 17, 22, 23]. It requires further investigation of how to augment them with aggregate weights for sampling purposes.

Using aggregate tree for uniform sampling was previously discussed in [14, 24] and it was discussed in [30] how to perform weighted sampling in an aggregate tree. In [19, 31], aggregate trees are used for drawing random samples from join queries. Aggregate trees may also be used for efficient range aggregation query [28]. None of these works consider concurrent updates and our work can help support concurrency in them.

One-dimensional independent range sampling, i.e., sampling from a key range instead of the entire table, in RAM model was studied in [11] and it supports drawing m samples in $O(logN + m)$ time and updates in $O(logN)$ time. However, it is not easy to implement in external memory and it is unclear how to support concurrent operations. AB-tree can be extended to support 1-D independent range sampling by only considering the index tuples that intersects the key range in Algorithm 1. The sampling time is $O(mlogN)$. which is higher than [11]. However, AB-tree's advantage is its high concurrency and support for external memory indexes.

7 CONCLUSION

In this work, we discuss the general design principles for concurrent aggregate B-tree and present AB-tree, an aggregate B-tree based on B-link tree that supports highly concurrent updates and random sampling operations. AB-tree can be a crucial component for supporting approximate query processing for real-time data analytics. Experiments show that our approach is more scalable and can achieve significantly higher throughput compared to the best-available baseline in a variety of workloads. That said, it is worth further investigation to alleviate the impact of the inherent contentions over the shared aggregates, especially there are many more cores and hyperthreading enabled in newer CPUs. We also discuss possible extensions of the design to other index structures and other sampling operations such as independent range sampling.

ACKNOWLEDGMENTS

Zhuoyue Zhao thanks the support for NSF IIS-1816149 and a Google PhD Fellowship. The authors appreciate the valuable feedback from the anonymous PVLDB reviewers for improving this manuscript.

REFERENCES

[1] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H. Chi, Lyric Pankaj Doshi, Tim Klas Kraska, Xiaozhou (Steve) Li, Andy Ly, and Chris Olston. 2020. Learned Indexes for a Google-scale Disk-based Database. In *ML for Systems workshop at NeurIPS '20*. https://arxiv.org/pdf/2012.12501.pdf

[2] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join Synopses for Approximate Query Answering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD '99)*. ACM, New York, NY, USA, 275–286. https://doi.org/10.1145/304182.304207

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*. 29–42.

[4] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 553–565. https://doi.org/10.1145/3164135.3164147

[5] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On Random Sampling over Joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/304182.304206

[6] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. 2016. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 679–694. https://doi.org/10.1145/2882903.2915249

[7] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-Free Linked Lists and Skip Lists. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing* (St. John's, Newfoundland, Canada) *(PODC '04)*. Association for Computing Machinery, New York, NY, USA, 50–59. https://doi.org/10.1145/1011767.1011776

[8] The PostgreSQL Global Development Group. 2021. PostgreSQL 13.1. https://www.postgresql.org/.

[9] Peter J. Haas. 1997. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *Ninth International Conference on Scientific and Statistical Database Management, Proceedings, August 11-13, 1997, Olympia, Washington, USA*, Yannis E. Ioannidis and David Marshall Hansen (Eds.). IEEE Computer Society, 51–63. https://doi.org/10.1109/SSDM.1997.621151

[10] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings (Lecture Notes in Computer Science)*, Jennifer L. Welch (Ed.), Vol. 2180. Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

[11] Xiaocheng Hu, Miao Qiao, and Yufei Tao. 2014. Independent Range Sampling. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Snowbird, Utah, USA) *(PODS '14)*. Association for Computing Machinery, New York, NY, USA, 246–255. https://doi.org/10.1145/2594538.2594545

[12] Ibrahim Jaluta, Seppo Sippu, and Eljas Soisalon-Soininen. 2005. Concurrency Control and Recovery for Balanced B-Link Trees. *The VLDB Journal* 14, 2 (April 2005), 257–277. https://doi.org/10.1007/s00778-004-0140-6

[13] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 631–646. https://doi.org/10.1145/2882903.2882940

[14] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., USA.

[15] Vladimir Lanin and Dennis Shasha. 1986. A Symmetric Concurrent B-Tree Algorithm. In *Proceedings of 1986 ACM Fall Joint Computer Conference* (Dallas, Texas, USA) *(ACM '86)*. IEEE Computer Society Press, Washington, DC, USA, 380–389.

[16] Philip L. Lehman and s. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 650–670. https://doi.org/10.1145/319628.319663

[17] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[18] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[19] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 615–629. https://doi.org/10.1145/2882903.2915235

[20] David Lomet and Betty Salzberg. 1992. Access Method Concurrency with Recovery. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (San Diego, California, USA) *(SIGMOD '92)*. Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/130283.130336

[21] David Lomet and Betty Salzberg. 1997. Concurrency and Recovery for Index Trees. *The VLDB Journal* 6, 3 (aug 1997), 224–240. https://doi.org/10.1007/s007780050042

[22] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[23] Maged M. Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Winnipeg, Manitoba, Canada) *(SPAA '02)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881

[24] F. Olken. 1993. *Random Sampling from Databases.* Ph.D. Dissertation. University of California at Berkeley.

[25] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1461–1476. https://doi.org/10.1145/3183713.3196905

[26] Yehoshua Sagiv. 1985. Concurrent Operations on B-Trees with Overtaking. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Portland, Oregon, USA) *(PODS '85)*. Association for Computing Machinery, New York, NY, USA, 28–37. https://doi.org/10.1145/325405.325409

[27] V. Srinivasan and Michael J. Carey. 1991. Performance of B-Tree Concurrency Control Algorithms. *SIGMOD Rec.* 20, 2 (April 1991), 416–425. https://doi.org/10.1145/119995.115860

[28] Jaideep Srivastava and Vincent Y. Lum. 1988. A Tree Based Access Method (TBSAM) for Fast Processing of Aggregate Queries. In *Proceedings of the Fourth International Conference on Data Engineering.* IEEE Computer Society, USA, 504–510.

[29] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 473–488. https://doi.org/10.1145/3183713.3196895

[30] C. K. Wong and Malcolm C. Easton. 1980. An Efficient Method for Weighted Sampling Without Replacement. *SIAM J. Comput.* 9, 1 (1980), 111–113. https://doi.org/10.1137/0209009

[31] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1525––1539. https://doi.org/10.1145/3183713.3183739