On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes

Yihan Sun Guy E. Blelloch Wan Shen Lim Andrew Pavlo
Carnegie Mellon University
{yihans, guyb, wanshenl, pavlo}@cs.cmu.edu

ABSTRACT

Modern data-driven applications require that databases support fast analytical queries while undergoing rapid updates—often referred to as Hybrid Transactional Analytical Processing (HTAP). Achieving fast queries and updates in a database management system (DBMS) is challenging since optimizations to improve analytical queries can cause overhead for updates. One solution is to use snapshot isolation (SI) for multi-version concurrency control (MVCC) to allow readers to make progress regardless of concurrent writers.

In this paper, we propose the Parallel Binary Tree (P-Tree) index structure to achieve SI and MVCC for multicore in-memory HTAP DBMSs. At their core, P-Trees are based on pure (immutable) data structures that use path-copying for updates for fast multi-versioning. They support tree nesting to improve OLAP performance while still allowing for efficient updates. The data structure also enables parallel algorithms for bulk operations on indexes and their underlying tables. We evaluate P-Trees on OLTP and OLAP benchmarks, and compare them with state-of-the-art data structures and DBMSs. Our experiments show that P-Trees outperform many concurrent data structures for the YCSB workload, and is 4–9× faster than existing DBMSs for analytical queries, while also achieving reasonable throughput for simultaneous transactional updates.

PVLDB Reference Format:
Yihan Sun, Guy Blelloch, Wan Shen Lim, Andrew Pavlo. On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes. *PVLDB*, 13(2): 211-225, 2019.
DOI: https://doi.org/10.14778/3364324.3364334

1. INTRODUCTION

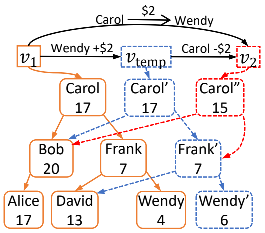
There are two major trends in modern data processing applications that make them distinct from database applications in previous decades [73]. The first is that analytical applications now require fast interactive response time to users. The second is that they are noted for their continuously changing data sets. This poses several challenges for supporting fast and correct queries in DBMSs. Foremost is that queries need to analyze the latest obtained data as quickly as possible. Data has immense value as soon as it is created, but that value can diminish over time. Thus, it is imperative that the queries access the newest data generated, without being blocked or delayed by ongoing updates or other queries. Secondly, the DBMS must guarantee that each query has a consistent view of the database. This requires that the DBMS atomically commit transactions efficiently in a non-destructive manner (i.e., maintaining existing versions for ongoing queries). Finally, both updates and queries need to be fast (e.g., exploiting parallelism or employing specific optimizations).

To address concurrent updates, one solution is to use multi-version concurrency control (MVCC) [87, 20, 67]. Instead of updating tuples in-place, with MVCC each write transaction creates a new version without affecting the old one so that readers accessing old versions still get correct results. In *snapshot isolation* (SI) every transaction sees only versions of tuples (the "snapshot") that were committed at the time that it started. Many DBMSs support SI, including both disk-oriented [58, 74] and in-memory [88, 68, 78] systems. The most common approach to implement SI is to use *version chains* [75, 18, 88], which maintains for each tuple a list of all its versions. A drawback of version chains, however, is that it can make readers slower: finding a tuple that is visible to a transaction requires following pointers and checking the visibility of each tuple version. One can reduce this overhead by maintaining additional metadata (e.g., HyPer creates *version synopses* [68]) about tuples, but those approaches have other overheads.

Modern DBMSs employ several approaches to accelerate read-heavy workloads but usually at the cost of slower updates. For example, columnstores allow for better locality and parallelism, such that queries accessing the same attribute within multiple tuples run faster [27]. However, it makes insertions and deletions more expensive [39], and also requires delicately-designed locking schemes that can inhibit certain updates to a tuple or version chain [19, 57, 72, 75]. Another way to improve OLAP performance is to denormalize tables or use materialized views to pre-compute intermediate results for frequently executed queries. Both of these approaches make updates more expensive because of the overhead of updating tuples in multiple locations [63] or invalidating the view.

To achieve high-performance in both updates and queries, we propose the **Parallel Binary Tree** (**P-Tree**) for multi-versioned, in-memory database storage. At their core, P-Trees are balanced binary trees that provide three important benefits for HTAP workloads on multi-core architectures. Foremost is that P-Trees are pure (immutable, functional) data structures (i.e., no operations modify existing data). Instead of version chains, P-Trees use path-copying, also referred to as copy-on-write (CoW), to create a new "copy" of the tree upon update. This means that the indices themselves are the version history without requiring auxiliary data structures—all data is accessed through the indices. Figure 1 presents an illustration of using path-copying for SI. Second, the trees use divide-and-conquer algorithms that parallelize bulk operations on tables—including filter, map, multi-insert, multi-delete, reduce, and range queries. These

Proceedings of the VLDB Endowment, Vol. 13, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3364324.3364334



- □ **Committing** v_2 as the latest version only requires to swing in v_2's root.
- □ **Rolling back** can be done by re-attaching the current version to v_1's root.
- □ The changes in the transaction (Carol -$2, Wendy +$2) are visible **atomically**.
- □ v_{temp} can be **collected**. Only the node Carol' will be freed because the other nodes are referenced by other trees.
- □ A transaction **reads** a version by acquiring the root pointer. An **update** transaction generate a new version by copying nodes on the affected path. A **read** or an **update** on a single tuple costs time $O(\log n)$.

Figure 1: P-Trees Multi-Versioning Overview – An example of using P-Trees to support a bank balance DBMS. The original state is v_1. A transaction transfers $2 from Carol to Wendy.

algorithms are based on using efficient operations that split and concatenate trees [23, 83], referred to as `concat`-based algorithms, and an efficient work-stealing scheduler for fork-join parallelism [26, 3]. Lastly, P-Trees support the *nesting of indexes*, one inside another. Such nested indexes improve OLAP query performance while still allowing for efficient updates under SI.

Using a pure data structure means that the DBMS must serialize updates to the global view or combine them together into batches. Some previous CoW-based systems, like LMDB [4], only allow for a single active writer and thus serialize writes. Others, like Hyder [21, 17, 76], support "melding" trees, but the melding process is still sequentialized, and furthermore can cause aborts. These limitations are the major bottleneck in CoW-based systems. P-Trees exploit parallelism by supporting parallel bulk operations as mentioned above. For any large transactions, or batched transactions consisting of multiple insertions and deletions, P-Trees can leverage multiple cores to atomically commit database modifications in parallel.

To evaluate P-Trees, we compare them with state-of-the-art concurrent data structures and in-memory DBMSs (MemSQL [78] and HyPer [68]). Our results show that on an OLTP workload (consisting of searches and updates) P-Trees are competitive with existing data structures that do not support multi-versioning. On an OLAP workload TPC-H, P-Trees are 4–9× faster than MemSQL and HyPer on a 72-core machine. On an HTAP benchmark containing updates with logging enabled, P-Trees remain almost as fast on the queries while supporting update rates comparable to what is supported by MemSQL (and much faster than HyPer). We also study what contributes to our performance gain of P-Trees compared to the other systems on queries, by removing some of the features. Our results show that much of the improvement is due to better parallel scaling (62.2× speedup on average using 72 cores) and the index nesting optimization (2× performance improvement on average).

Our contributions can be summarized as follows.

1. The combination of path copying and parallel bulk operations in an MVCC database is new. The bulk parallelism leads to very good speedup and performance on both queries and batch updates while supporting full serializability.
2. The use of nested indexes based on nesting trees, along with their application to fast queries. This leads to significant speedup on many of the TPC-H queries.
3. The C++ implementation of a DBMS based on P-Trees. This includes the implementation of nested indexes, support for parallel bulk operations, and an implementation of all 22 TPC-H queries. We have made the code available on GitHub [6].
4. An experimental evaluation comparing P-Trees to other DBMSs to analyze its effectiveness.
5. A new benchmark that adds TPC-C style transactions to TPC-H. This differs from the CH-benCHmark [37] , which adds TPC-H style queries to TPC-C.

We note that although some of the contributions seem independent (e.g., bulk operations, nested indexes), an important aspect of the work is that P-Trees make it easy to combine these ideas.

2. BACKGROUND

We first present some background information to help motivate design of the P-Tree.

Transaction Isolation Levels. A transaction's isolation level determines what anomalies it may be exposed to during its execution. These were originally defined in the context of pessimistic two-phase locking concurrency control in the 1990s. *Snapshot isolation* (SI) is an additional level that was proposed later after the original standard was released [16]. SI is a popular isolation level and is often good enough for HTAP environments because its OLAP queries will be read-only. Marking an OLAP transaction as read-only means that the database does not need to maintain its read-write set while it executes. All transactions still check the visibility of each tuple. A DBMS is serializable if its outcome of executing any concurrent transactions is equal to executing its transactions in a serial order.

Concat-based Algorithms. We employ an algorithmic framework for balanced binary trees based on a subroutine $\texttt{concat}(T_L, e, T_R)$. This function takes two balanced binary search trees T_L and T_R and a node e as input. We call e the ***pivot*** of this `concat` function. All keys in T_L need to be smaller than the key of e, which must be smaller than all keys in T_R. It returns a balanced tree containing all entries in T_L, T_R as well as e. In other words, this function returns a tree that is equivalent to concatenating T_L and T_R with e in the middle, but with balancing issues resolved. The pivot e also can be empty, in which case the algorithm just concatenates the two trees. Previous work [23] discussed parallel `concat`-based algorithms[1].

Pure Data Structures and Path-copying. A data structure is considered *pure* if its internal structure is never modified even when the contents are updated. Each update on the data structure will create a new copy. To make this operation more efficient, the data structure shares common substructures across copies. With trees, this is usually achieved by *path-copying*—i.e., copying the affected path to the update. Such path-copying is the standard approach in functional languages since the early days of Lisp (the 1960s), and is also used in some previous multi-version data structures [80, 13] and disk-based database systems [4, 9, 21, 76, 17]. An algorithm is considered pure if it does not cause side-effect on its inputs.

The PAM Library. All of our implementations are built on top of the PAM library [83]. PAM is a parallel library supporting the pure `concat`-based algorithms on trees by path-copying.

Parallel Cost Model. To analyze the asymptotic costs, we use work and span (depth), where work is the total number of operations and span is the length of the critical path. Any computation with W work and S span will run in time $T < \frac{W}{P} + S$ on P cores assuming shared memory and a greedy scheduler [50, 30, 25].

3. P-TREES: PARALLEL BINARY TREES

In this section, we describe our P-Tree data structure. We will first present an overview of P-Trees. We will then discuss parallel bulk algorithms for updates and queries. All algorithms in this section are pure using path-copying. We show all code in Figure 2. In the pseudocode presented in this section, `add_ref` increments the reference counter of a tree node. $\texttt{copy}(x)$ creates a tree node with the same data as x. `lc` and `rc` indicate left and right child pointers.

[1] The primitive `concat` was referred to as `join` in the original paper. In this paper we use `concat` to avoid confusion with the `join` operation in relational algebra.

```
1   node* multi_insert(t, A, m, $\rho$) {
2     (A2, m2) = parallel_sort_and_combine(A, m, $\rho$);
3     return multi_ins_s(t, A2, m2, $\rho$);}
4   node* multi_ins_s(t, A, m, $\rho$) {
5     if (t is null) return build(A, m);
6     if (m is 0) { add_ref(t); return t;}
7     int b = binary_search(A, m, t->key);
8     bool d = (b < m) and (A[b].key > t->key);
9     node* L = multi_ins_s(t->lc, A, b, $\rho$); ||
10      node* R = multi_ins_s(t->rc, A+b-d, m-b-d, $\rho$);
11    node t2 = copy(t);
12    if (d is 1) t2->value = $\rho$(t->value, A[b].value);
13    return concat(L, t2, R); }
```

(a) Multi-insertion

```
1   node* range(t, kl, kr) {
2     node* cur = t;
3     while (kl > cur->key or kr < cur->key) {
4       if (kl > cur->key) cur = cur->rc;
5       if (kr < cur->key) cur = cur->lc;}
6     node* L=right(cur->lc, kl), R=left(cur->rc, kr);
7     node mid = copy(cur);
8     return concat(L, mid, R); }
9   node* right(t, k) { // left(t, k) is symmetric
10    if (t is null) return t;
11    if (k > t->key) return right(t->rc, k);
12    node t2 = copy(t); add_ref(t->rc);
13    return concat(right(t->lc, k), t2, t->rc); }
```

(b) Range

```
1   node* filter(t, $\phi$) {
2     if (t is null) return null;
3     node* L = filter(t->lc, $\phi$); ||
4       node* R = filter(t->rc, $\phi$);
5     if ($\phi$(t)) return concat(L, copy(t), R);
6     else return concat(L, null, R);}
```

(c) Filter

```
1   VType map_reduce(t, $f_m$, $\langle f_r$, I$\rangle$) {
2     if (t is null) return I;
3     VType L = map_reduce(t->lc, $f_m$, $\langle f_r$, I$\rangle$); ||
4       VType R = map_reduce(t->rc, $f_m$, $\langle f_r$, I$\rangle$);
5     return $f_r$($f_r$(L, $f_m$(t->entry)), R); }
```

(d) Map-Reduce

```
1   void foreach_index(t, $\phi$, s) {
2     if (t is null) return;
3     int left = size(t->lc);
4     VType L = foreach_index(t->lc, $\phi$, s); ||
5       VType R = foreach_index(t->rc, $\phi$, s+1+left);
6     $\phi$(t->entry, left); }
```

(e) Foreach Index

Figure 2: Parallel Bulk Operations – Algorithms for parallel bulk operations on P-Trees. $S_1||S_2$ indicates that the two statements S_1 and S_2 can run in parallel.

P-Trees are balanced binary search trees that maintain a sorted set of *key-value* pairs. We allow the key and value to be of any type, such as integers, strings, or even another tree. Each tree node stores a key-value pair, two child pointers, the subtree size, and a *reference counter* for garbage collection (GC). The reference counter of a tree node records the number of references (e.g., the child pointer from a parent node, the handle to a root, etc.) pointing to it. We provide more details of the GC procedure in Section 6.

3.1 Parallel Bulk Update Operations

We use `multi_insert` and `multi_delete` to commit a batch of write operations. The function `multi_insert`(t, A, m, ρ) takes as input a P-Tree root t, the head pointer of an array A with its length m, and a combine function ρ. The combine function $\rho : V \times V \mapsto V$, where V is the value type of the tree is used to deal with duplicate keys. When inserting a key-value pair $\langle k, v\rangle$, if k is already in t, then the tree updates its value by combining the value of k in t with v using ρ. This is useful, for example, when the value is the accumulated weight of the key and the combine function is addition ($\rho(a, b) = a + b$). Another use example is when installing new values for existing keys ($\rho(a, b) = b$).

We present the pseudocode of `multi_insert` in Figure 2a. This algorithm first sorts A by keys, and then removes duplicates by combining all values of the same key using ρ. We then use a divide-and-conquer algorithm `multi_ins_s` to insert the sorted array into the tree. The base case is when either the array A or t is empty. Otherwise, the algorithm uses a binary search to locate t's key in the array, retrieving the corresponding index b in A. d is a bit that denotes whether k appears in A. Then the algorithm recursively multi-inserts A's left part (up to $A[b]$) into the left subtree, and A's right part into the right subtree. The two recursive calls can run in parallel. The algorithm concatenates the two results with a copy of the current node t (with its value updated if d is true).

Note that after Line 10, L and R are not necessarily balanced since the distribution of the input array can be arbitrary. However, the balance of the output tree is guaranteed as long as a valid `concat` algorithm is used. The rest of the algorithm need not rebalance the tree. This property also holds for the other algorithms in this section, which all use `concat` as a primitive for connecting and rebalancing.

We use a similar algorithm for `multi_delete`. The work and span of inserting or deleting an array of length m into a tree of size $n \geq m$ is $O\left(m \log\left(\frac{n}{m} + 1\right)\right)$ and $O(\log m \log n)$, respectively [23]. A DBMS uses these bulk update algorithms to commit a batch of operations. We will discuss more details in Section 5.

3.2 Parallel Bulk Analytical Operations

To facilitate read-only analytical queries, P-Trees support several analytical primitives. This is useful for maintaining intermediate results of primitives in the same representation as the input (i.e., another index structure) since this allows primitive cascading; this is also known as *index spooling* in Microsoft SQL Server [2]. The P-Trees primitives extract such intermediate views on the current snapshot and output another tree structure. This not only avoids additional data scans, but is also asymptotically more efficient than scanning the data directly (e.g., for the `range` function, see details below). The code of all algorithms is shown in Figure 2.

Range. This operator extracts a subset of tuples in a certain key range from a P-Tree, and output them in a new P-Tree. The cost of the `range` function is $O(\log n)$. The pure `range` algorithm copies nodes on two paths, one to each end of the range, and uses them as pivots to `concat` the subtrees back. When the extracted range is large, this pure `range` algorithm is much more efficient (logarithmic time) than visiting the whole range and copying it. Note that although intuitively extracting a range would take time proportional to the range size, our method avoids doing so by outputting a *tree*, and thus avoids touching output data—only the affected path is read and copied, instead of the whole range of entries. This is useful, for example, in cascading queries when the output is for further queries.

Filter. The `filter`(t, f) function returns a P-Tree with all tuples in t satisfying a predicate $f : E \mapsto$ `Bool` on the entry. This algorithm filters the two subtrees recursively, in parallel, and then determines if the root satisfies f. If so, the algorithm copies the root, and uses it as the pivot to `concat` the two recursive results. Otherwise it calls `concat` without a pivot. The work of `filter` is $O(n)$ and the depth is $O(\log^2 n)$ where n is the tree size. The pure version of `filter` leaves the original tree intact, and creates a new tree as the output.

Map-Reduce. The function map_reduce$(t, f_m, \langle f_r, I\rangle)$ on a tree t (with tuple type E) takes three arguments and returns a value of type V'. $f_m : E \mapsto V'$ is a map function that converts each tuple to a value of type V'. $\langle f_r, I\rangle$ is a monoid where $f_r : V' \times V' \mapsto V'$ is an associative reduce function on V', and $I \in V'$ is the identity of f_r. The algorithm recursively calls the function on its two subtrees in parallel, and reduces the results using f_r afterwards.

Foreach Index. foreach_index (t, ϕ, s) applies a function $\phi(e, i)$ to each tuple e in the tree rooted at t, and an integer $i = s + k$, where k is the index (starting from 0) of e in t. s is an offset that shifts the index. Because the index of the root of t is the size of t's left subtree, we first directly apply ϕ on t's root and offset s+size(t->lc). Then we recursively deal with the two subtrees in parallel. For the left subtree, the new offset is still set to be s, but the right subtree has the new offset s+size(t->lc)+1. This function is useful, for example, when we want to output all entries in a tree t to an array a, for which we can use:

```
  foreach_index(t, [](E e, int i) {a[i]=e;});
```

4. NESTED INDEXES

To support efficient analytical queries, the P-Tree can use *nested* and *paired* indexes. A nested index embeds one index inside another such that each value of the top level is itself another index. A paired index uses a single index for two tables that share the key, often a primary key in one and a secondary or foreign in the other. We show an example in Figure 3 on the TPC-H workload, and will explain it in detail later in this section. Both nested and paired indexes can be considered a "virtual" denormalization of the data. In particular, a paired index is logically a pre-join on the shared key, and a nested index roughly corresponds to adding a pointer from the parent to the child and indexing on it (sometimes referred to as the *short-circuit key*). The nesting and pairing does not materialize the view—there is no copying of the tuples, and therefore it suffers less from the problems of additional space, consistency, and expensive updates. We will discuss the space overhead of index nesting in Section 6.

Nesting and pairing is straightforward with P-Trees since it supports arbitrary key and value types. Furthermore, nested and paired tables work well with both its parallel operations and with path copying. These operations over a nested index, such as map and reduce, can themselves be nested so there is parallelism both on the outer index and inner index (we provide an example below). Path copying works with nesting since the path from an outer tree continues into a path on the inner nested tree (see Section 5).

In general, one can apply nesting across multiple levels. For example, for TPC-H we create up to three levels (e.g., customer-order-lineitem, or part-supplier-lineitem). We also use nested indexes for indexes on secondary keys—the outer index is on the secondary key and each inner index contains the tuples with the same secondary key indexed by their primary key. This differs from the more common implementation of secondary indexes that keeps inner sets of elements with the same secondary keys as lists or arrays [49]. Using a nested index has the advantage of being able to quickly find and delete an element in the inner index based on its primary key.

We are not aware of any DBMS that uses index nesting based on trees. This is possibly because such nesting is unlikely to be efficient on disk-based DBMSs due to fine grained memory accesses, and because it is complicated to to combine nesting with existing optimization techniques, such as column stores or version chaining. P-Trees make such nesting easy to support, and we do use a columnar data model or version chaining. The general idea of index nesting does not rely on TPC-H or the P-Tree, and thus is of independent interest and can be extended to other settings.

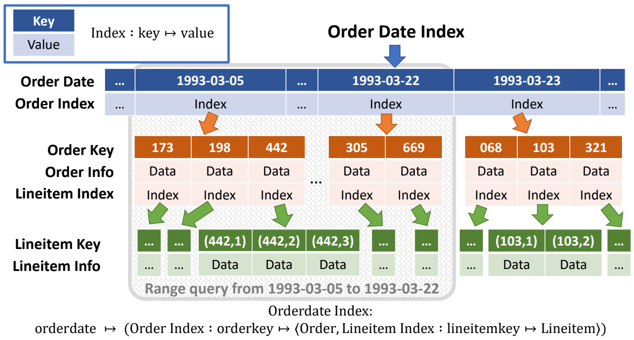


Figure 3: Nested Tree Structure for TPC-H – An example of a three-level nested P-Tree for the orderdate-order-lineitem relation in TPC-H. A range query on the top level index would effectively filter out irrelevant lineitems in the bottom level, making queries more efficient.

To illustrate how nesting and pairing works, we consider an example from TPC-H shown in Figure 3. The top level index for the ORDER table is on the non-unique orderdate key (i.e., the date on which a customer made an order). Within each date, there is an index of the orders on that date keyed on the primary key orderkey. In the TPC-H schema, each order has a set of items called lineitems, or more precisely, each lineitem has an orderkey as part of its two-attribute primary key. Since lineitems share the orderkey with orders, we can pair the indexes. This pairing is shown by the two orange tuples for "Order Info" and "Lineitem Index". The order info contains the complete table entry for each order. This is either (1) the tuple directly stored in the tree node or (2) a pointer to the tuple elsewhere in memory. In our experiments, we compare the two methods. Importantly, the example shows a third level of nesting on the lineitems themselves, indexed based on their primary keys, which consists of both the orderkey and the linenumber. Each of the inner most indexes is itself represented as a P-Tree, even though in TPC-H there are at most seven lineitems per order. The example shows both three-level nesting and pairing.

This nested index can be thought of as a virtual pre-join of the ORDER and LINEITEM tables on their shared orderkey, and then indexing the result based on orderdate. As the illustration shows, however, the orders are not copied across multiple lineitem tuples, and the join is never materialized. We note the same index can be used to answer range queries by date on just orders, or on the lineitems that belong to a range of orderdates.

To show why the nesting and pairing are useful, we consider TPC-H Q4. We consider both how existing DBMSs process the query and then how the nested-paired index can be used to significantly improve performance. The query in SQL is given in Figure 4a. The query looks up the orders where (1) the orderdate is in a given range and (2) there exists a lineitem for the order such that commitdate < receiptdate. It then counts the number of relevant orders for each different orderpriority (i.e., five). This query accesses both the ORDER and LINEITEM tables. But DBMSs often scan the LINEITEM table first, which is problematic because there are 4× as many tuples as the the total number of orders, and 28× as many as the number of orders in the orderdate range. We discuss this problem further in Sections 8.3 and 8.4 for MemSQL and HyPer.

We now consider how to perform the query using the nested index illustrated in Figure 3. The DBMS can first do a range search on a specific date range, identifying the ⟨order, lineitem index⟩ pairs that fall within the range (i.e., the orders in the shaded rectangle in Figure 3). For TPC-H Q4, this is about 1/28 of all the orders. Then

```sql
SELECT o_orderpriority, COUNT(*) AS order_count
  FROM orders
 WHERE o_orderdate>=date '[DATE]'
   AND o_orderdate<date '[DATE]'+interval '3' month
   AND EXISTS ( SELECT * FROM lineitem
                 WHERE l_orderkey=o_orderkey
                   AND l_commitdate<l_receiptdate )
 GROUP BY o_orderpriority
 ORDER BY o_orderpriority;
```

(a) SQL

```cpp
1  using Arr = Array<int, NUM_PRI>; //NUM_PRI is 5
2  Arr Q4(DB d, const Date q) {
3    odate_tree t = d.odate_idx.range(q, q+month(3));
4    auto date_f = [] (odate_entry& dt) -> Arr {
5      auto ord_f = [] (order_entry& o) -> Arr {
6        auto item_f = [] (Lineitem& l) -> bool {
7          return l.c_date<l.r_date;};
8        int p=o.get_order().orderpriority()-1;
9        Arr a;
10       a[p]=o.item_idx.map_reduce(item_f,OR());
11       return a; };
12     return dt.ord_idx.map_reduce(ord_f,Add_Arr());
13   };
14   return t.map_reduce(date_f, Add_Arr()); }
```

(b) P-Tree Implementation

Figure 4: TPC-H Q4 – The definition TPC-H Q4 and the pseudocode of using map-reduce functions on P-Trees for implementing TPC-H Q4. *OR* in the code means the logical-OR operation on boolean values.

for each such order it can examine all the lineitems that belong to the order to see if any satisfy the predicate on order and commit date. If so, then the system increments a counter for the appropriate order priority, which it can combine with a parallel reduce.

Our code for Q4 using P-Trees is given in Figure 4b. It extracts the range of order dates in Line 3. It then has nested parallel calls to `date_f` (over `orderdates`), then `ord_f` (over all orders for a given date), and finally `item_f` (over all lineitems of a given order). The outer two both use `map_reduce` and the inner-most just checks if the lineitem satisfies `commit_date < receipt_date`. This producs an approximately 10 fold improvement over MemSQL, HyPer, and our own DBMS without nested indexes (see Section 8.3). We also added the secondary index on `orderdate` in HyPer, but it did not help since the system still needed to scan the `LINEITEM` table.

4.1 Defining Nested Indexes

In our DBMS, we supply a simple way for users to construct nested indexes by building them up based on three primitives. All the indexes, both the outer and inner ones, are maintained by P-Trees. To construct a nested index, we define the following functions.

- **`primary(Table, primarykey)`**: Construct an index from a table `Table` based on the `primarykey`.
- **`secondary(Index, secondarykey)`**: Construct a secondary index from a primary index `Index`, based on the `secondarykey`. If there are multiple tuples sharing the same secondary key, build inner indexes based on the primary key.
- **`pairing(Index1, Index2)`**: For two indexes `Index1`: $X \mapsto Y$ and `Index2`: $X \mapsto Z$, construct an index that maps X to a pair of Y and Z. This is similar to a regular `join` operation, but keep the join column as the key of the output index.

These three primitives fully support the index nesting we propose in this paper. We have implemented the three functions based on P-Trees and use them in our experiments. Here we show an example of defining the nested indexes in the example of Figure 3 using these three primitives.

```
     I0 = primary(LINEITEM, lineitemkey)
     I1 = primary(ORDER, orderkey)
     I2 = secondary(I0, orderkey)
  I_ORD = pairing(I1, I2)
I_ODATE = secondary(I_ORD, orderdate)
```

The first two lines build primary indexes on lineitems and orders, the third line then builds a secondary index for the lineitems based on the `orderkey`, and the fourth line pairs up the primary order index `I2` with this secondary lineitem index `I3`. The last line indexes the paired index based on the secondary key `orderdate`, and returns a three-level index `I_ORD`.

As another example of building a three level index, the following will construct a customer index on top of the `I_ORD` index.

```
    I3 = primary(CUSTOMER, custkey)
    I4 = secondary(I_ORD, custkey)
I_CUST = pairing(I3, I4)
```

It is a three-level nested index, each level also is a paired index with customer and order tuples stored in it, respectively. This index is used frequently in our implementation of TPC-H queries. In queries that require a `join` on lineitems and a selection of customers (e.g., Q3), we can filter out the irrelevant customers on the top level such that their lineitems will not be scanned. In TPC-H, there are 40× more lineitems than customers, so the total accessed data will be much less than all the lineitems. The improvement of using a nested index in Q3 is more than 300% (see Table 3).

5. USING PURE P-TREES FOR SI

In this section, we describe how to use pure P-Trees for SI and MVCC, and how to achieve serializability using batching. P-Trees use pure `concat`-based algorithms that never modify existing tree nodes, but copy necessary parts when updates occur. As a result, multiple logical versions of indexes share physical tree nodes.

MVCC Example. We use the example in Figure 1 to show how pure P-Trees support MVCC and SI. In P-Trees, a transaction acquires the current version of an index by grabbing the root and incrementing the root's reference counter (see Section 6) to create a snapshot of the index. When our system must maintain multiple consistent indexes, it keeps a top-level tuple storing a pointer to each index. We refer to this as the "world" since it stores all dynamic information for a database. Acquiring a version grabs a pointer to this tuple. Creating a new world on updating an index requires copying this tuple and putting in the new root of the updated index. The tuple (the world) is only a constant number of pointers.

The DBMS creates local versions of indexes using analytical operations, such as `range` and `filter`, but it does not need to commit these local versions. Any updates also create new versions, but the DBMS commits them by writing a pointer to the new world. Figure 1 shows an example of a transaction on a single index. The transaction transferring $2 from Carol to Wendy is implemented by two consecutive updates using path-copying, leading to a new version v_2. Committing the new version v_2 involves updating the current root pointer to v_2's root. Then the two updates in the transaction become visible atomically. Such atomic updates are applicable to either a series of updates like in this example, or a batch of updates (e.g., by a pure `multi_insert`). The old versions are still available and can be accessible by the old root pointers, so ongoing queries can continue working on them. Such a strategy also makes rollbacks easy by swinging back the old root pointer. The DBMS's GC algorithm can collect old versions (see Section 6).

Using P-Trees for MVCC requires no additional versioning or other internal auxiliary fields in the data structure. The DBMS can

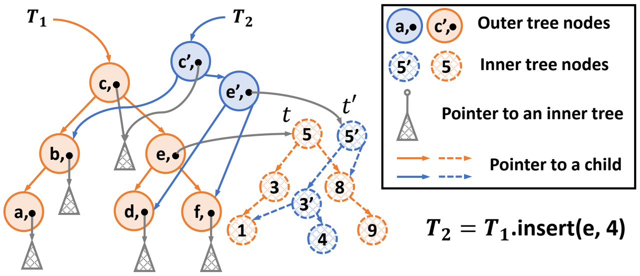


Figure 5: Functional Update on a Multi-level P-Tree – The update uses path-copying on both inner and outer trees.

maintain versioning information (e.g., timestamps and liveness), with the set of root pointers to the versions.

Pure Update on Nested Trees. As described in Section 4, P-Tree's index nesting accelerates the analytical queries. Using P-Trees to store nested indexes has the benefit that updates are inexpensive, as the DBMS can perform the update by path-copying across both the outer and inner trees. Figure 5 shows an illustration of updating a two-level P-Tree. The update algorithm first finds the affected tree node e in the outer tree and then copies the path along the way. For the other copied nodes in the outer tree other than e, the inner trees do not change, and thus we can directly use the pointer to the original inner trees. For example, node c' in T_2 has the same inner tree pointer as c in T_1. For e itself, we copy it to e', and the algorithm then inserts 4 to the inner tree t of e (the orange slashed nodes), giving a new inner tree t'. The root of t' is then assigned to e' as the inner tree. The total cost of such an insertion is $O(\log n_I + \log n_O)$, where n_I and n_O are the sizes of the inner and outer trees respectively. As such, SI is still supported, and all algorithms in Section 3 remain applicable on a nested tree, or on multiple nested trees stored in a world.

5.1 Serializable Updates

One concern with SI is in supporting serializability for concurrent update transactions [34]. This is further complicated in path-copying-based (pure) approaches since each `update` makes its own copies of paths, which then need to be resolved if they run concurrently. A simple solution is to only allow a single writer to sequentialize all `updates` (e.g., flat-combining [51]). This is likely to be adequate if `updates` are large with significant internal parallelism, or if the rate of smaller `updates` is light (in the order of tens of thousands per second), as might be the case for a DBMS dominated by analytical queries. It is unlikely to be adequate for DBMSs with high update rates of small transactions. Hyder [76], which uses multiversioning with path copying, addresses this by allowing transactions to proceed concurrently and then merging the copied trees using a "meld" operation. The DBMS, however, must sequentialize these melding steps so that it creates new versions one at a time. The melding process can also cause an abort.

Another approach is to batch updates as part of a group commit operation [41]. The basic approach is for the DBMS to process a set of updates obeying a linear order. It then detects any logical conflicts based on this order and the operations they performed (e.g., write-read conflicts). The system next removes any conflicted updates and then commits the remaining conflict-free updates as a batch. This approach is taken by a variety of systems, including Calvin [84], FaunaDB [1], PALM [77], and BATCHER [7]. The advantage of this approach is that the batch update can make use of parallelism [7, 1], consensus is only needed at the granularity of batches in distributed systems [84, 1], and the batched updates can make more efficient use of the disk in disk-based systems [10]. The challenge of the approach, however, is in detecting conflicts; all of the above DBMSs perform this step differently.

In our system we use batching and use multi-insert and multi-delete to apply batches of point updates. As discussed, these primitives have significant internal parallelism. In this paper we do not consider how to detect conflicts for arbitrary transactions (previous work could help here), but study the approach for simple point transactions such as in the YCSB benchmark (insertions and deletions). In this case, the only conflicts are between updates on the same key, and keeping the last update on the key is sufficient. We note it is also easy to detect conflicts in the TPC-C transactions we evaluate: for the New-Order transactions described in Section 8.1, one can check if two transactions share a customer or part-supplier. In our experiments, our DBMS simply sequentializes them because the workload is comprised of mostly analytical queries.

To batch a set of updates, we wait for some amount of time, allowing any updates to accumulate in a buffer, and then apply all the updates in the buffer together as a batch in parallel. While processing the batch new updates can accumulate in another buffer. In this approach there is a tradeoff between throughput and latency—the longer we wait (higher latency), the better the throughput due to increased parallelism. This tradeoff is analyzed in Section 8.2.

In summary, based on batching, our solution allows for parallel group commit for concurrent point updates. This in fact provides lock-free write transactions (insertions/deletions/updates) and wait-free read-only transactions (e.g., find and range queries). For more general and complicated transactions that involves more than one operations, analyzing the dependency among transactions is necessary for achieving parallelism. One can also simply serialize all transactions when the update load is not heavy.

6. IMPLEMENTATION DETAILS

In this section, we provide implementation details of P-Trees.

Parallelism. The core aspect of the P-Tree's implementation is its use of fork-join parallelism with a dynamic scheduler. We also use the parallel aggregation and sorting algorithms from the PBBS library [24]. The parallelism on P-Trees mainly comes from the divide-and-conquer scheme over the tree structure. As we will show in Section 8, P-Trees achieve almost linear scalability.

To control the granularity of the parallel tasks, the P-Tree only executes two recursive calls in parallel when the current tree size is larger than some threshold. Otherwise the tree processes the operations sequentially. This is to avoid the overhead of forking and joining small tasks. This granularity level is adjustable and is decided by the workload of the base case dealing with a single tree node. Intuitively, if the base cases are light-loaded, we make it more coarse-grained. For example, by default we stop generating parallel tasks when the tree size is under 100. Otherwise, when the work within a single node is sufficient (e.g., they deal with inner trees), parallelism can be introduced even to the bottom level of the tree.

Memory Management and GC. The P-Tree's memory manager maintains separate pools of nodes for each thread, along with a shared lock-free stack-based pool. The tree maintains blocks of 64k nodes in the shared pool. Each thread retrieves a block from the shared pool when they run out of nodes in their local pool, and then returns a block when they accumulate 2×64k free tree nodes.

In each P-Tree, the data structure shares tree nodes across snapshots to bound the time and space. Such node sharing, however, requires a carefully designed garbage collector to avoid deleting visible nodes or retaining unreachable nodes indefinitely. For P-Trees we implement a reference counting garbage collector (GC) [54]. Each tree node will maintain a *reference counter* (RC), which records the number of references (pointers) to it, e.g., child pointers from parent nodes, handles to a root, etc. The system uses an atomic fetch-and-add to update RCs since multiple threads can potentially update an RC concurrently. We use the standard reuse optimization where if a node has a reference count of one (only the current call has a pointer to it), then it is reused rather than being copied [54]. This allows in-place updates when possible.

To identify which nodes can be reclaimed, we use an approach that is similar to the one proposed in [15]. When the DBMS deletes a snapshot, the GC decrements the RC of its root. Whenever a node's counter reaches zero, our GC frees the node and returns it to the owning thread's local pool. The GC then reclaims the node's two children recursively in parallel. Note that the GC collects all the tree nodes that are *only* in this collected snapshot. This does not affect any other tree nodes in other snapshots except for decrementing their reference counters when necessary.

Durability. We make our DBMS durable by writing logs of transactions to the disk in our experiments. We note that because P-Trees support snapshot isolation, it is also feasible to write the snapshot of the DBMS in a certain frequency, and only write logs in between these snapshots, making reconstruction of a certain version cheaper.

7. SPACE OVERHEAD

Here we discuss the space overhead of our approach from index nesting and path-copying, respectively.

Space Overhead from Index Nesting. P-Tree's nesting method is similar to data denormalization and materialized views, but avoids copying of data across rows. This both saves memory and reduces the cost and complexity of updates. However, there is some space cost of index nesting. In particular, each nested index in which a table row can appear can cause a copy of that row. In our TPC-H implementation, for example, the lineitems are copied four times in different nested indices because each lineitem is involved in several hierarchies and secondary indexes. The copy can either involve storing the row directly in a tree node, or a pointer from the tree node to a shared copy of the row. The second approach requires less memory since the row is shared among indices only requiring a pointer within each index. It may require extra time in analytical queries due to a level of indirection and the additional cache misses this incurs. One can also save space by using fewer secondary indexes, which also slows down some queries.

Space Overhead from Path-copying. Using path-copying inherently copies more data than other MVCC solutions based on version chains since it copies the whole path to the update rather than just the affected tuple itself. However, it needs less metadata within the data structures (e.g. timestamps on each version within a version list). It also can easily garbage collect any old versions, which is complicated with version chains. To be more concrete, to insert or delete a batch of m tuples into a tree of size $n \geq m$, the number of extra tree nodes created by our `multi_insert` algorithm is $O\left(m\left(\log\frac{n}{m}+1\right)\right)$ [23]. This may seem high, but it is always asymptotically bounded by n (the size of the original index). In practice it is usually small since m is much less than n. However, if many versions are kept, this cost can accumulate over the versions if they are not collected. In Section 8.5 we experimentally evaluate this memory overhead as a function of the number of versions.

8. EXPERIMENTAL EVALUATION

We now provide a comprehensive evaluation of P-Trees under a variety of settings and workload conditions. In our implementation, P-Trees are the sole data representation; the DBMS accesses tuples through them. For all of these experiments, we use a 72-core Dell R930 with four Intel Xeon E7-8867v4 (18 cores, 2.4GHz, and 45 MB L3 cache), and 1 TB memory. Each core is two-way hyperthreaded giving a total of 144 hyperthreads. Our code was compiled using g++ 5.4.1. We use `numactl` in the experiments with more than one thread to spread the memory pages across CPUs in a round-robin fashion.

We first compare P-Trees with four concurrent data structures on OLTP workloads. We then compare our system to two DBMSs HyPer [56] and MemSQL [78] on both an OLAP and HTAP workloads. Finally we analyze the space overhead of our system from path-copying and index nesting.

8.1 Workloads

We first describe the benchmarks we use in our evaluation. For our P-Tree, we implement the benchmarks in our testbed DBMS. For the SQL-based systems (HyPer and MemSQL), we use the open-source OLTP-Bench benchmarking framework [44].

YCSB. We test YCSB workloads **Insert-only** (constructing the tree with parallel insertions), **A** (read/update, 50/50), **B** (read/update, 95/5), and **C** (read-only) with Zipfian distributions. The database contains a single table with 50m entries using 64-bit integers as keys and values. Each workload contains 10^7 transactions. Our DBMS executes read transactions concurrently on the last committed snapshot in the database. We buffer the write transactions using the batching algorithm in [14, 15] and then commit them to the last snapshot of the database using `multi_insert` without blocking readers. We execute batched updates with the `replace` combine function (see Section 3). If there are multiple updates on the same key in a batch, then the last write persists. We report the mean of throughput across five trials and the standard deviation.

As described in Section 3.1, batching performance is highly affected by the batch granularity. On the one hand, we need to make batches reasonably small and frequent such that each query get a timely response. On the other hand, larger and less frequent batches, although causes long latency, usually lead to less overhead and better throughput. As a result, the appropriate batching size achieves a tradeoff between latency and throughput, which depends on the platform and the machine. We will show some in-depth study of the correlation between latency and throughput. Except for the experiments on latency-throughput tradeoff, which varies the latency, we control the latency to be 50 ms, which is the same magnitude of network latency, and thus is unlikely to dominate the cost. As a result, 50 ms is the acceptable limit in many application domains (e.g., internet advertising, on-line gaming).

TPC-H. We use this OLAP workload to evaluate the performance of our DBMS executing analytical queries. We report the geometric mean of the running time for each query across five trials, and also report the geometric mean of all 22 queries, as is suggested in the TPC-H official document [5]. Of the eight tables in the TPC-H database, we use seven nested P-Tree structures (labeled with *) to maintain primary and secondary indexes. The exact configuration of the database is formalized in Figure 6. To represent a nested index, we use the value type as an index represented by another tree. We also keep four arrays for static data on the `SUPPLIER`, `PART`, `NATION`, and `REGION` that map the primary key to the corresponding tuple. We note that none of the nested indexes created in our DBMS is just for a specific query. For example, a total of six different TPC-H queries take advantage of the three-level customer index T_{cust}.

TPC-HC. To provide a more in-depth analysis of hybrid workloads, we created a hybrid benchmark called TPC-HC based on TPC-H and TPC-C. Unlike the HyPer's CH-benCHmark [38], which

Inner indexes:

$$\begin{aligned}
T_{\text{lineitem}} &= \mathbb{T} : \langle \text{orderkey, linenumber} \rangle \mapsto \text{lineitem} \\
T_{\text{partsupp}} &= \mathbb{T} : \langle \text{partkey, suppkey} \rangle \mapsto \langle \text{partsupp}, T_{\text{lineitem}} \rangle
\end{aligned}$$

Primary indexes:

$$\begin{aligned}
T_{\text{order}} &= \mathbb{T} : \text{orderkey} \mapsto \langle \text{order}, T_{\text{lineitem}} \rangle && (*) \\
T_{\text{cust}} &= \mathbb{T} : \text{custkey} \mapsto \langle \text{customer}, T_{\text{order}} \rangle && (*) \\
T_{\text{supp}} &= \mathbb{T} : \text{suppkey} \mapsto \langle \text{supplier}, T_{\text{partsupp}} \rangle && (*) \\
T_{\text{part}} &= \mathbb{T} : \text{partkey} \mapsto \langle \text{part}, T_{\text{partsupp}} \rangle && (*)
\end{aligned}$$

Secondary indexes:

$$\begin{aligned}
T_{\text{receiptdate}} &= \mathbb{T} : \text{date} \mapsto T_{\text{lineitem}} && (*) \\
T_{\text{orderdate}} &= \mathbb{T} : \text{date} \mapsto T_{\text{order}} && (*) \\
T_{\text{shipdate}} &= \mathbb{T} : \text{date} \mapsto T_{\text{lineitem}} && (*)
\end{aligned}$$

Figure 6: Maintaining TPC-H Tables with Nested P-Trees – $\mathbb{T} : K \mapsto V$ denotes a tree storing a mapping from K to V. $\langle A, B \rangle$ means a pair of two elements A and B. The first two indexes defined below are used only as inner trees. The indexes tagged with "*" are the outer trees used in our implementation.

Table 1: Updates on Trees in TPC-HC – The trees that require updates in each transaction type.

Txn	Table	Operations
New-Order	T_{cust},T_{order} $T_{\text{orderdate}}$, T_{supp}, T_{part}, Q_{order}	• Add to the following relations: partsupp ↦ lineitem, customer ↦ order, order ↦ lineitem, orderdate ↦ order • Decrease each item's available quantity • Enqueue this order to Q_{order}
Payment	T_{cust}	• Decrease the customer's balance
Delivery	Q_{order}, T_{shipdate} T_{part}, T_{supp} T_{cust}, T_{order} $T_{\text{orderdate}}$,	• Dequeue some orders from Q_{order} • Update lineitems' shipdate, customers' balance, lineitems' and orders' status, • Add to the shipdate ↦ lineitem relation

integrates TPC-H queries into TPC-C, our benchmark integrates TPC-C transactions into TPC-H workloads. We do this because our DBMS is optimized for OLAP queries, and thus we want to provide a more fair comparison with other systems on TPC-H queries.

This benchmark contains all 22 queries from TPC-H along with the following three transactions derived from TPC-C with their denoted percentage of the total update workload:

- **New-Order [49%]**. A customer creates a new order entry that contains $1 \leq x \leq 7$ lineitems.
- **Payment [47%]**. A customer makes a payment to update their account balance.
- **Delivery [4%]**. The company marks the lineitem records in the earliest $1 \leq y \leq 10$ un-processed orders as shipped.

The order and lineitem information are generated based on TPC-H specification. For our implementation, the required actions of each transaction on our TPC-H configuration is shown in Table 1. During each trial, the system uses one thread for the update transactions and another thread to invoke TPC-H queries. We use the same code as in our TPC-H experiments and run queries in parallel with all available threads. All of the OLTP transactions and OLAP queries operate on the latest snapshot available to them. After each OLTP transaction finishes, the DBMS commits their updates atomically. They are then immediately visible to the next TPC-H query.

All updates for P-Tree are running sequentially. We allow running multiple update transactions at the same time for MemSQL.

8.2 Data Structure Comparison

In this first experiment, we compare P-Trees with four other state-of-the-art concurrent indexes for in-memory DBMSs:

- **B+tree**. A memory-optimized B+tree using optimistic locking coupling [61].
- **Bw-Tree**. Microsoft's latch-free B+tree index from Hekaton [62]. We use the OpenBw-Tree implementation [86].
- **MassTree**. A hybrid B+tree that uses tries for nodes from the Silo DBMS [66].
- **Chromatic Tree**. A lock-free Chromatic tree implementation in C++ [33, 31, 69].

The implementation of B+tree, OpenBw-Tree and MassTree are from Wang et al. [86, 85]. The Chromatic Tree implementation is from Brown et al. [32, 33]. We note that none of them support SI.

We use three experiments to evaluate P-Tree's performance on the OLTP benchmark YCSB. We first compare the throughput of all tested data structures on four workloads to understand the parallel performance of these data structures under different read/write ratios. We then experiment on the tradeoff between latency and throughput for P-Trees, using workload A as an example. Finally, we discuss the scalability curve for all tested data structures on all four workloads.

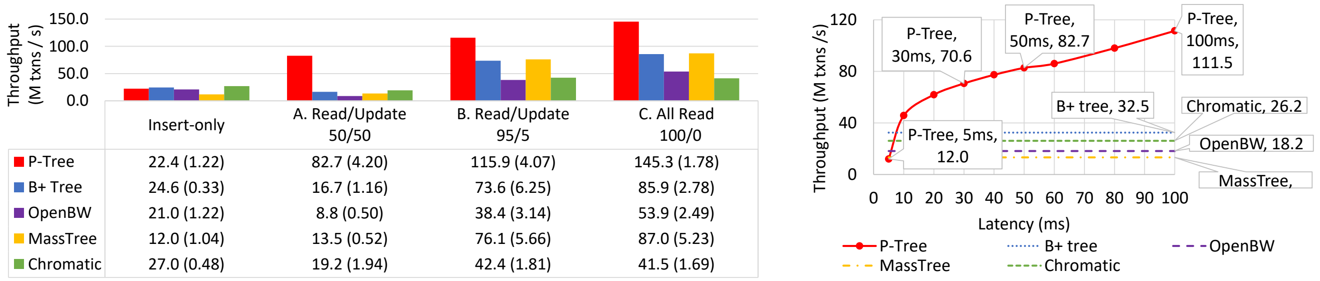
Performance. We first measure the throughput of all data structures. The numbers reported on P-Trees are within 50 ms latency. Figure 7a shows that P-Trees outperform or is competitive to the other implementations. P-Trees' standard deviation is within 6% in all test cases. P-Trees' throughput improves as the ratio of reads increases. This is because each read transaction on the P-Tree operates on a snapshot of the tree and is not blocked by other transactions.

For the Insert-only workload, P-Trees outperform OpenBw and MassTree, but is 10% slower than B+tree, and 20% slower than Chromatic tree. Chromatic trees' high update throughput is at the expense of low read performance (workload C). B+trees' better performance is likely to come from shallow height and better cache locality. On the other three workloads, P-Trees generally demonstrate better performance than the other data structures. The main overhead in our DBMS is the overhead of batching since our DBMS has to buffer all the transactions' operations. Meanwhile, P-Trees's good performance comes from better parallelism and non-contention achieved by batching. We will discuss more details later.

Latency vs. Throughput To better understand how batching affects the performance of P-Trees, we next measure the system's performance when varying the acceptable latency, which is done by adjusting the batch size and waiting time between batches. We control the 99% of longest time (P99 latency) each transaction waits for a response, and test the throughput of P-Trees. We then compare this against the best performance of the other data structures measured in Figure 7a for any thread count configuration. We first load 50m entries into the database and then use 144 threads to execute the YCSB Workload A transactions.

The results in Figure 7b show that, to match the best of the other four data structures, P-Trees only causes $\sim$10 ms latency. At a 50 ms latency window, P-Trees are more than 2.2× faster than all the other indexes. With more strict restriction in latency (e.g., when only 5 ms latency is allowed), the P-Tree's performance is worse than all the other tested data structures).

Scalability. Lastly, we measure how the indexes perform as we scale up the number of concurrent threads. We first observe that the results in Figure 8a show that most of the data structures are able to scale on the insert-only workload. For the mixed workloads, Figures 8b to 8d show that P-Trees achieve good scalability and parallelism, and is the only index that scales up to 144 threads in all workloads. The other four indexes suffer from bad performance with more threads in workload A (50/50 reads/updates). This is expected as more concurrent threads create more contention in the data structure. P-Trees avoid this contention by allowing reads to access an isolated tree structure. For write transactions, our

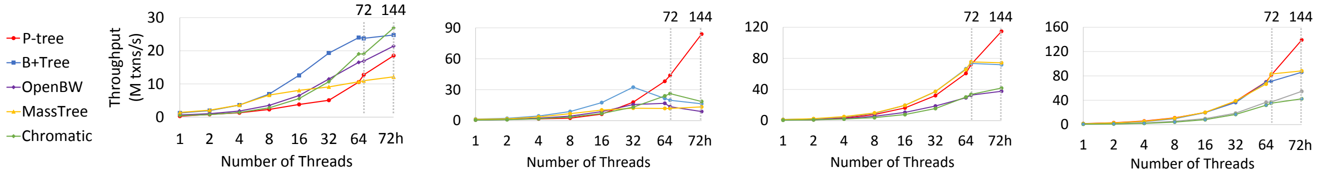


	Insert-only	A. Read/Update 50/50	B. Read/Update 95/5	C. All Read 100/0
🟥 P-Tree	22.4 (1.22)	82.7 (4.20)	115.9 (4.07)	145.3 (1.78)
🟦 B+ Tree	24.6 (0.33)	16.7 (1.16)	73.6 (6.25)	85.9 (2.78)
🟪 OpenBW	21.0 (1.22)	8.8 (0.50)	38.4 (3.14)	53.9 (2.49)
🟨 MassTree	12.0 (1.04)	13.5 (0.52)	76.1 (5.66)	87.0 (5.23)
🟩 Chromatic	27.0 (0.48)	19.2 (1.94)	42.4 (1.81)	41.5 (1.69)

(a) Throughput

(b) Throughput-Latency Correlation (Workload A)

Figure 7: YCSB Workload Performance – Comparison of concurrent data structures for the YCSB workloads using 144 threads. (7a) Throughput measurements for each data structure. Numbers in the parentheses are the standard deviations. P-Trees execute concurrent updates in batches within 50 ms latency. (7b) The correlation between latency and throughput on YCSB workload A. The horizontal lines show the maximum throughput of the other data structures across different numbers of threads.



(a) Insert-only **(b)** Workload A (50/50 Read/Update) **(c)** Workload B (95/5 Read/Update) **(d)** Workload C (Read-only)

Figure 8: YCSB Workload Scalability – We always control the latency to be ∼50 ms for P-Trees. "72h" means 144 threads (with hyperthreading).

implementation batches and executes them in a parallel divide-and-conquer algorithm. This also avoids contention because no two threads work on the same tree node at the same time.

8.3 OLAP Workload Evaluation

For this next group of experiments, we test P-Trees on all 22 queries in TPC-H and compare it with two in-memory DBMSs:

- **MemSQL (v6.7):** We load all of the TPC-H tables into the DBMS's column-store engine. We then use the `memsql-optimize` tool to configure the DBMS's runtime components. Although we ran our experiments on a single node, MemSQL does not support databases with a single partition, where a partition is MemSQL's unit of query parallelism. Thus, we use the `memsql-ops` tool to configure the DBMS with two partitions per CPU.
- **HyPer (v20181.18):** We use the commercial version of the HyPer binary shipped in the Tableau distribution. We use DBMS's default runtime configuration that sets inter-query and intra-query parallelism to use all available cores. For sequential time testing, we limited HyPer's inter-query and intra-query parallelism to one core each.

For both MemSQL and HyPer there are many settings and we made a significant effort to run the workload in the best possible way. This included contacting the developers of both and using the configurations suggested by them for the TPC-H workload. The numbers we report are comparable to those which are self reported for the two database systems [35, 60] on TPC-H and adjusted for processor capabilities.

We note that MemSQL's column-store engine does not support additional indexes. For HyPer we ran with and without secondary indexes and took the best time. Our implementation with P-Tree uses secondary indexes on `orderdate`, `shipdate`, and `receiptdate`. P-Trees do not use optimizations based on column stores (everything is a row store), NUMA optimizations, or vectorization.

We note that both HyPer and MemSQL are full-featured DBMSs, which have more functionalities than our DBMS, including SQL compilation and durability. To make a more fair comparison, we exclude the query compilation time for them and also make our DBMS durable by writing logs of transactions to disk.

We use TPC-H scale factor (SF) 100 and run all tests five times to achieve more stable results. The results are presented in Table 2 in the columns labeled with "✗" in updates. The columns labeled with "seq." show the sequential running time, and the columns with "par." are parallel running time. We use grey cells to note the best throughput number among the three systems.

Parallel Performance. All three systems achieve stable query performance across runs. The geometric standard deviation of P-Trees, HyPer, and MemSQL are 1.015, 1.023 and 1.037, respectively. In all 22 queries, P-Trees are faster than both HyPer and MemSQL. P-Trees are at least 3× faster on most queries compared to both MemSQL and HyPer. However, in some queries (e.g., Q1, Q6, Q8, Q12 and Q21) where P-Trees cannot take advantage of using nested indexes, P-Trees' advantage is less significant (only 1-2× faster). For example, Q1 and Q6 requires scanning several specific columns of almost all lineitems, and thus nested indexes cannot help to pre-filter. Q8 involves two nestings (customer-order-lineitem and part-partsupp-lineitem), and thus using hierarchical map-reduce function on one index still need an extra lookup at the parent-child relation in the other index. Q12 and Q21 are similar. On the other hand, in these queries an implementation could benefit from using columnstore by reducing I/O, as both HyPer and MemSQL do. In fact, the good performance of P-Trees mainly comes from two aspects: better parallelism, and the index nesting. We will discuss this in more detail later in this section.

The geometric mean of all 22 queries using P-Trees is 4.7× faster than HyPer and 9.6× faster than MemSQL.

Sequential Performance and Scalability. Overall, the P-Tree's performance is still better than HyPer and MemSQL, but not as significant as for parallel performance. For the geometric mean of the sequential running time of all 22 queries, the P-Tree is about 2.3× faster than HyPer and 6× faster than MemSQL. Similarly, in the queries (e.g., Q6, Q8, Q12 and Q21) where P-Trees cannot take much advantage of nested indexes, P-Trees's performance can be similar or even slower than HyPer, because the advantage of columnstore is more significant. However, P-Trees still achieve better parallel performance in these queries, indicating that our implementation potentially allows for better scalability.

Table 2: TPC-H Measurements (SF 100) – "seq." means sequential running time in seconds. "par" means (intra-query) parallel running time in milliseconds. ✓ or ✗ means with or without updates running at the same time. Gmean=geometric mean across five runs. Running time in the last row shows the geometric mean of the each column. The mean of ratios are calculated from the mean of the running time. For parallel runs, we use all 144 threads. We highlight (the grey cells) the highest throughput numbers among the three implementations.

	MemSQL					**HyPer**					**P-Tree**				
Parallel?	**Par. (ms)**			**Seq. (s)**	**speed-**	**Par. (ms)**			**Seq. (s)**	**speed-**	**Par. (ms)**			**Seq. (s)**	**speed-**
Update?	✗	✓	**ratio**	✗	**up**	✗	✓	**ratio**	✗	**up**	✗	✓	**ratio**	✗	**up**
Q1	375	436	16.3%	11.6	31.0	354	345	-2.5%	12.4	35.0	324	331	2.0%	6.6	20.5
Q2	233	295	26.6%	9.1	38.9	255	256	0.4%	9.1	35.7	15	16	9.5%	0.8	51.9
Q3	2377	2494	4.9%	12.2	5.1	441	469	6.3%	14.1	31.9	144	147	1.9%	9.4	65.6
Q4	403	504	25.1%	27.9	69.2	357	386	8.1%	14.6	40.9	36	37	3.6%	3.1	87.9
Q5	1171	1174	0.3%	36.3	31.0	507	543	7.1%	17.7	34.9	68	70	1.7%	5.0	72.4
Q6	230	310	34.8%	7.3	31.7	100	103	3.0%	1.1	11.2	51	52	0.4%	2.7	53.6
Q7	579	904	56.1%	13.2	22.8	381	393	3.1%	12.1	31.8	60	62	3.5%	4.4	72.6
Q8	298	335	12.4%	15.1	50.8	125	137	9.6%	4.8	38.6	94	96	2.7%	5.8	61.9
Q9	1726	1915	11.0%	127.9	74.1	1176	1200	2.0%	47.8	40.7	184	184	0.1%	7.9	42.8
Q10	700	808	15.4%	66.9	95.6	404	385	-4.7%	11.4	28.2	53	55	2.4%	4.2	78.4
Q11	120	124	3.3%	1.9	15.7	67	81	20.9%	1.4	20.7	14	14	2.6%	0.7	53.4
Q12	277	378	36.5%	10.2	36.9	120	121	0.8%	4.6	38.2	105	106	1.0%	10.2	97.2
Q13	4561	4033	-11.6%	279.5	61.3	1559	1645	5.5%	73.4	47.1	406	414	2.0%	34.3	84.4
Q14	250	244	-2.4%	14.7	58.8	79	249	215.2%	7.0	88.5	22	25	13.3%	1.6	71.0
Q15	1131	1795	58.7%	26.4	23.3	204	221	8.3%	3.0	14.7	25	27	7.9%	1.3	49.6
Q16	660	730	10.6%	35.8	54.2	426	436	2.3%	7.2	16.8	70	73	3.6%	5.7	81.1
Q17	258	265	2.7%	8.1	31.3	261	285	9.2%	7.2	27.7	36	37	5.2%	1.2	34.9
Q18	6327	6762	6.9%	43.1	6.8	3135	3484	11.1%	37.7	12.0	425	428	0.8%	29.3	68.9
Q19	180	208	15.6%	10.9	60.4	222	240	8.1%	8.5	38.3	28	30	5.7%	2.2	77.4
Q20	1737	2012	15.8%	95.0	54.7	192	335	74.5%	9.0	47.0	25	26	2.7%	2.3	90.6
Q21	1333	1402	5.2%	363.2	272.5	798	858	7.5%	30.5	38.2	263	276	5.1%	15.5	59.1
Q22	613	640	4.4%	26.5	43.2	181	187	3.3%	6.1	33.9	39	42	8.8%	2.5	65.0
Mean	640.5	734.3	14.6%	24.7	38.5	311.3	352.7	13.3%	9.6	30.8	66.4	68.9	3.9%	4.1	62.2

Using all 72 cores with hyperthreading, P-Trees achieve a 62× speedup on average, while HyPer and MemSQL achieve around 30×. This indicates that P-Trees' performance greatly benefits from better scalability.

Parallel Performance Gain Breakdown. To understand the performance gain of our TPC-H implementation, we look at three optimizations we use: the index nesting, the secondary indexes, and whether we include the table entries *inline* in the index (as with index organized tables [81]), or in a separate record elsewhere in memory with a pointer to it. We implement 12 representative queries in TPC-H, that use different tables and indexes in our implementation. We start from plain P-Trees with no optimization, and add the three optimizations one by one to test the improvement. Results are shown in Table 3. We mark the test cases that P-Trees are better than *both* HyPer and MemSQL as grey cells.

For data inline, the performance gain is about 12%. This means that storing pointers in indexes to save memory overhead caused by index nesting causes about 12% overhead in queries. Even using plain P-Trees, our implementation outperforms both HyPer and MemSQL in four out of 12 tested queries. Five out of 12 tested queries achieve improvement from secondary indexes. The improvement ranges from 24% to 1400%. Even without index nesting, P-Trees are better than both HyPer and MemSQL in nine out of 12 queries. Eight out of 12 tested queries take advantage of tree nesting. The improvement of using index nesting is 0.9-10×.

In summary, the major query performance gain of P-Trees' is from index nesting. Some queries (e.g., Q1) require scanning all lineitems, which cannot benefit from index nesting. In this case, P-Trees with all optimizations can only achieve similar performance to a plain P-Tree. For the others, the overall improvement of all optimizations can be up to 20× faster than the plain P-Trees.

8.4 HTAP Workload Evaluation

We test P-Trees on the hybrid benchmark TPC-HC (see Section 8.1). We run our system with durability enabled, and terminate after 10^5 transactions. We show the running time in Table 2. Queries are in the columns labeled with ✓. We compare our results to HyPer and MemSQL.

Table 3: P-Tree TPC-H Performance Breakdown (SF=100) – The running time for 144 threads with different optimizations enabled, including index nesting (Nested), secondary index (Sec. Idx) and data inline (Inline). "Ratio" means the improvement compared with the previous column. We highlight (the grey cells) all P-Trees' throughput numbers that outperform *both* HyPer and MemSQL.

	P-Tree	**P-Tree**		**P-Tree**		**P-Tree**		**Hyper**	**MemSQL**
Nested	no	no		no		**yes**		-	-
Sec. Idx	no	no		**yes**		yes		**yes**	-
Inline	no	**yes**		yes		yes		-	-
	time	**time**	**ratio**	**time**	**ratio**	**time**	**ratio**	**time**	**time**
Q1	388	317	22.3%	324	-2.3%	324	0.0%	334	375
Q2	59	52	12.9%	52	0.0%	15	257.5%	251	233
Q3	1053	985	6.9%	985	0.0%	144	584.5%	464	2377
Q4	630	517	22.0%	417	24.0%	36	1068.6%	363	403
Q5	613	531	15.5%	444	19.4%	68	548.9%	520	1171
Q6	395	325	21.8%	51	533.6%	51	0.0%	99	230
Q8	1044	984	6.2%	984	0.0%	94	949.9%	125	298
Q11	49	52	-6.1%	52	0.0%	14	278.6%	67	120
Q12	505	472	6.9%	105	349.1%	105	0.0%	120	277
Q13	794	777	2.1%	777	0.0%	406	91.4%	1595	4561
Q14	409	337	21.4%	22	1399.7%	22	0.0%	82	250
Q18	1998	1718	16.3%	1718	0.0%	425	304.4%	3174	6327
Gmean	446	398	12.0%	233	70.8%	77	201.8%	286	585

Table 4: Update Throughput on TPC-HC – Updates are executed sequentially, while queries are executed one by one using multi-cores. *: HyPer's performance is below expectation. More explanation is given in Section 8.4.

	New-Order	**Payment**	**Delivery**	**Overall**
MemSQL	23,655	25,156	765	10,280
HyPer*	67	214	11	75
P-Tree	7,037	61,332	1,110	8,696

The throughput numbers of HyPer are not satisfactory, which are 100× slower than MemSQL and our P-Tree, and are much slower than they report in their paper [56] (we note that the benchmark is slightly different, and our workload is 100× larger than that in [56]). Through correspondence with the original HyPer authors, we understand the Tableau version does not have certain features enabled. We suspect that this contributes to the degradation of performance observed.

Update Overhead. As shown in Table 2, for P-Trees, adding a sequential update process only causes less than 5% overhead to the TPC-H queries, which is lower than both HyPer and MemSQL.

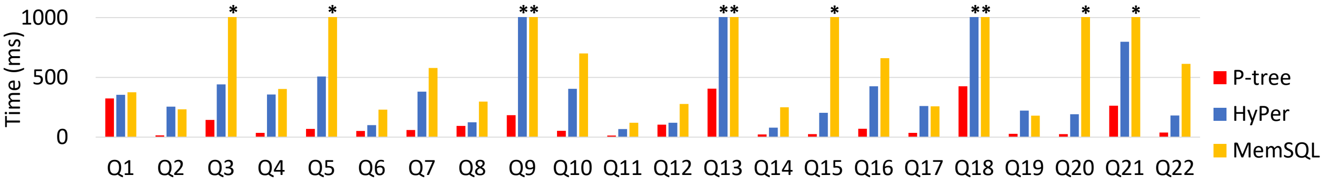


Figure 9: Parallel TPC-H Running Time (SF=100) – The parallel (144 threads, time in milliseconds) running time of TPC-H queries. We cut of the y-axis at one second. "*" in the figure indicates that those numbers exceed the y-axis.

Table 5: Running time of updates with GC – k versions are batched and collected in parallel. $k = +\infty$ means no GC.

k	10	50	100	$+\infty$
Time (s)	15.67	7.09	6.49	5.87

The slowdown is likely caused by the contention in updating the reference counters, which will invalidate cache lines. We note that there are a small number of queries that have higher throughput when running with updates. This is possible that when running with updates, the query coincidentally gets better cache locality.

P-Tree Update Throughput. Our DBMS achieves a throughput of 12k update transactions per second on a single thread. For the Payment transactions on P-Trees, the only updated tree is the index for customers. Thus the throughput is the highest among the three. Each New-Order transaction updates five trees. Therefore it is much slower (about 8×) than the Payment transactions. Finally the Delivery transaction is the most expensive among the three, not only because it needs to update an average of five orders, but it also updates six trees in total. Therefore, it is about 5× slower than the New-Order transactions.

Update Throughput Comparison with MemSQL. Overall P-Trees have almost exactly the same performance as MemSQL in updates. On New-Order it is about 3× slower, while on Payment it is about 3× faster, and about 2× faster on Delivery. The extra cost on New-Order is due to the extra tables that have to be updated.

GC Cost. We test the GC cost of P-Trees for 10^5 updates. After each k transactions, we collect the latest k versions in parallel. Results are shown in Table 5. When GC is performed frequently, the cost can be comparable to the update cost because collecting a version also goes through a path in the tree. When more versions are buffered and collected in parallel, the GC cost gets smaller. When $k = 100$, the GC overhead is only about 10%. Buffering 100 versions only requires less than 1 GB extra memory (see Section 8.5).

8.5 Memory Overhead Analysis

In this section, we analyze the memory overhead caused by path-copying and index nesting in our TPC-H and TPC-HC experiments.

Index Nesting Memory Overhead. Index nesting can cause extra space when a table is involved in multiple logical hierarchies or secondary indexes. One can expect space-performance tradeoff by designing different nestings of indexes. We now analyze the memory usage in our TPC-H nested index implementation.

In total, to store 100 GB TPC-H raw data, our implementation uses 265 GB data, including the raw data, metadata for indexes (e.g., pointers and subtree sizes for each node), and duplicates caused by index nesting. This number matches the theoretical estimation of memory usage in Table 6 in the second last column *Inline*. This gives us the good performance as we report in Table 2. One can save space by storing only pointers in indexes, avoiding physically copying data. For our TPC-H implementation, this saves about 55 GB data, as we show in the last column *Indirect* in Table 6. However using indirections may cause extra cache misses for queries. This overhead in query time is about 12% as we show in Table 3. We can also save space by storing fewer indexes. For example, if we drop the secondary index on `receiptdate`, we can save about 30 GB memory. The only side effect is the slowing down of Q12 by 4×.

Both HyPer and MemSQL use ~100 GB memory on 100 GB TPC-H raw data. We do not use any compression as they do. We note that there are also ways to compress P-Trees [42].

Path-copying Memory Overhead. We next measure the memory overhead for maintaining multiple snapshots using P-Trees through path-copying. We present our results in Figure 10. We execute 1.5×10^6 New-Order transactions (1% of the original database size). For simplicity, we only evaluate two indexes in our system: the customer-order-lineitem nested index T_{cust}, and the part-partsupp-lineitem nested index T_{part}. We keep k (ranging from 15 to 1.5M) recent versions, and garbage collect other versions. We present the peak memory footprint over time in Figure 10. The grey dotted line shows the actual memory size required by the new orders. We do not count the memory to store strings because they are not stored in the indexes, and we never copy them. The extra used memory includes the added data itself, metadata to maintain the trees (e.g., pointers in each node) as well as extra space caused by path-copying.

To add a new order O_n from customer C_n in T_{cust}, we first build a tree of all lineitems in O_n, attach it to a new created order node, and insert this node into the inner tree of C_n. We use nested copying in Section 5. Each lineitem creates exactly one lineitem node because they are not inserted into existing trees, but a newly-built tree. Therefore there is no extra copying of lineitem nodes. Orders are stored in small inner trees of the customers. Therefore the overhead in copying order nodes is also small because each new order only creates about 3–4 order nodes. The major overhead occurs in copying customer nodes. Every new order copies a path of customers. However, when k is small, the out-of-date customer nodes all get collected due to precise garbage collection, leaving only small memory overhead (less than 0.02 GB). In fact, the used space is almost exactly $k \cdot h_c \cdot s_c$, where $h_c \approx 23$ is customer tree height, and $s_c = 64B$ is the customer node size. If more versions are kept, the memory overhead is more than 4×.

In T_{part}, every new lineitem inserted leads to path-copying on all three levels of nesting, causing more overhead than T_{cust}. The order data are not added to this index. The total memory overhead is generally low when old versions are collected in time. Similarly, the lineitem and partsupp trees are all shallow inner trees. Thus the overhead mainly comes from a large number of copied part nodes especially when k is large.

In both cases, index nesting helps to reduce memory overhead from path-copying because the dominating memory usage, which are the lineitems, are kept only in small shallow trees. In fact, index nesting shifts the copying of inner tree nodes to outer tree nodes. In our indexes, many of the outer tree nodes have to be updated and copied anyway because of the New-Order transactions (e.g., the available quantity of a partsupp). Therefore the total number of copied nodes reduces because of index nesting.

In summary, the space overhead of P-Tree is small when GC is in time or when the change of the DBMS is not significant, but

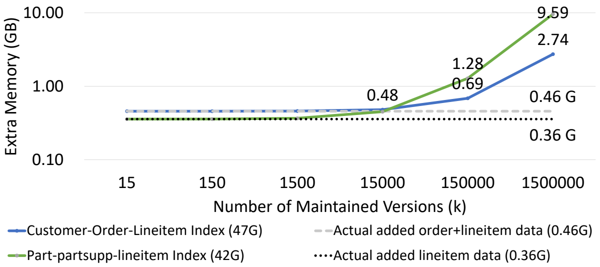


Figure 10: P-Tree TPC-HC Memory Overhead (SF=100) – The memory overhead of executing 1.5M New-Order transactions for the TPC-H benchmark. The x-axis shows the numbers of kept versions k. The original size of customer-order-lineitem is ∼47 GB, and the original size of the part-partsupp-lineitem index is ∼42 GB.

will become a big bottleneck if more versions remain in memory. Usually, it is unlikely to have the number of living versions be more than the number of physical threads. In our case, the overhead of keeping 144 versions is rather low. If we want to keep all history, we can also write old versions to disk to collect them in memory.

9. RELATED WORK

MVCC and Snapshot Isolation. MVCC [87, 20, 67] is a widely-used technique in various DBMSs for allowing fast and correct read transactions without blocking (or being blocked by) the writers. MVCC is implemented in many DBMSs [43, 59, 46, 79, 74, 68]. Some of these existing systems, like Hekaton [59] and HyPer [68], use timestamp-based version chains for concurrency control. This may require the read transactions to scan the version chain and check the visibility of the version at the current timestamp, which can be expensive when there are a large number of versions. HyPer uses *version synopses* to avoid expensive scans, but this does not have any theoretical guarantee in bounding the reading time. P-Trees avoid this by using the functional data structure.

Copy-on-write and Functional Data Structures. Copy-on-write (CoW) means to explicitly copy some resource if and only if it is modified. Indeed path-copying is a specific implementation of CoW. Similar ideas are also used in *shadow paging* [64] to guarantee atomicity. Path-copying has been used in maintaining multiversion B-tree or B+tree structures or their variants [80, 13]. Path copying is the default implementation in functional languages, where data cannot be overwritten [70]. It is widely-used in real-world database systems for version-controlling like LMDB [4], CouchDB [9], Hyder [21] and InnoDB [47], as well as many other systems [36, 28, 40, 52, 55]. Most of these only allow a single update at a time, sequentializing the writes. Hyder supports merging of path copied trees. This means that transactions can update trees concurrently, and then each is merged into the main trees one at a time. This allows for some pipelining [17] but still fully sequentializes the commits at the root (only one merge can complete at a time). It also means that transactions can abort during the merge if they have any conflicts. Importantly these other systems have focused on point updates (single insertions, deletions, or value changes). An important aspect of P-Trees is that they support parallel bulk updates on pure trees. This allows for batching of transactions in certain situations and processing them in parallel to get much higher throughput.

`concat`*-based Algorithms.* Blelloch et al. studied `concat`-based algorithms for parallel set-set algorithms [23], augmentations [83], computational geometry [82], and multi-version concurrency [15].

Query Optimizations. Our proposed index nesting is related to many previous techniques proposed to support high throughput of OLAP queries. As mentioned, the paired index enables virtually denormalization [53] of two tables. The main challenging of physical denormalization is the high memory assumption, and the high cost to update while maintaining SI. There are attempts aiming at reducing the space required by data denormalization using compression [63]. P-Trees avoid these costs because we never physically copy data multiple times by making use of the nested trees.

Table 6: P-Tree TPC-H Memory Usage (SF=100) – "String" is the size of string per entry, "Data" means the size of other fields per entry. Strings are not copied in duplicates.

	Bytes per entry			Dup-	Size	Memory (GB)	
	String	Metadata	Data	licates	(M)	Inline	Indirect
Lineitem	79	24	40	4	600	187.20	138.02
Order	109	40	32	3	150	45.40	39.81
Customer	207	40	24	1	15	3.79	3.90
Partsupp	199	40	24	2	80	24.36	23.77
Part	148	40	24	1	20	3.95	4.10
Supplier	181	40	24	1	1	0.23	0.24
Total						**265 GB**	**210 GB**

Although our nested indexes usually provide a view of pre-join over tables, our approach differs from materialized views [48] in that we do not physically create a new table, such that the update is still cheap using path-copying.

Table partitioning means to partition a table in storage to represent the parent-child relation [8, 45, 12]. Table partitioning was also previously employed to manage TPC-H workload, especially to manage the belongness between orders and lineitems. Our index nesting conceptually uses the same idea of putting all lineitems in the same order together, but differs in that we build an index using a tree inside. Also, these previous work on table partitioning focusing more on partitioning the table for distributed systems. The idea of representing relations between objects across tables in a graphical or hierarchical way is similar to the Resource Description Framework (RDF) [71, 29, 11] and path-indexing [22, 65] in Object-Oriented Databases (OOD). Our index nesting differs from them in that we usually only allow for regular tree or DAG nesting, instead of arbitrary graphic relations. Also, we propose to use nested pure tree structures to support fast queries and updates on such indexes.

10. CONCLUSION

In this paper, we present P-Trees, a functional tree structure for supporting a multi-version DBMS with SI, that allows safe and efficient concurrent updates and queries. In P-Trees, we employ parallel bulk operations for better parallelism, nested indexes to improve the performance for OLAP queries, and path-copying for supporting SI and MVCC. As a result, P-Trees demonstrate good performance in both updates and queries in various hybrid workloads.

On an OLTP workload YCSB, P-Trees outperform or are competitive to several existing concurrent data structures, and demonstrate good scalability in multi-core systems. On an OLAP workload TPC-H, P-Trees are 4-9× faster than existing DBMSs. On an HTAP workload, P-Trees remain up to 9× faster, and achieve close performance in updates. The good query performance of the P-Tree mainly comes from (1) good parallelism, as P-Trees achieve on average 62× parallel speedup on TPC-H, and (2) the optimization of index nesting, as the performance of the P-Tree improves by more than 180% by using index nesting.

Acknowledgments

This work was supported (in part) by the National Science Foundation (SPX-1822933, XPS-1629444, AF-1910030), Google Research Grants, and the Alfred P. Sloan Research Fellowship program.

11. REFERENCES

[1] FaunaDB. https://fauna.com/.

[2] Index spooling in microsoft sql server. https://sqlserverfast.com/epr/index-spool/.

[3] The problem based benchmark suite (PBBS) library. https://github.com/cmuparlay/pbbslib.

[4] Lightning memory-mapped database manager (LMDB). http://www.lmdb.tech/doc/, 2015.

[5] TPC benckmark™H standard specification revision 2.18.0. 2018.

[6] The PAM library, an HTAP database system on TPC workloads, 2019.

[7] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 84–95, 2014.

[8] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2004.

[9] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide: Time to Relax*. O'Reilly Media, Inc., 2010.

[10] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, 2002.

[11] M. Athanassoulis, B. Bhattacharjee, M. Canim, and K. A. Ross. Path processing using solid state storage. In *Proceedings of the 3rd International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2012)*, number CONF, 2012.

[12] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1165–1176. ACM, 2011.

[13] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[14] N. Ben-David, G. Blelloch, Y. Sun, and Y. Wei. Efficient single writer concurrency. In *arXiv preprint arXiv:1803.08617*, 2018.

[15] N. Ben-David, G. Blelloch, Y. Sun, and Y. Wei. Multiversion concurrency with bounded delay and precise garbage collection. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2019.

[16] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[17] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1295–1309. ACM, 2015.

[18] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[19] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[20] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[21] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *Innovative Data Systems Research (CIDR)*, volume 11, pages 9–20, 2011.

[22] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on knowledge and data engineering*, 1(2):196–214, 1989.

[23] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2016.

[24] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 47, pages 181–192, 2012.

[25] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1):202–229, 1998.

[26] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[27] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[28] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. In *Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[29] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2013.

[30] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, Apr. 1974.

[31] T. Brown. Lock-free chromatic trees in c++. `https://bitbucket.org/trbot86/implementations/src/`, 2016.

[32] T. Brown. Lock-free chromatic trees in c++. `https://bitbucket.org/trbot86/implementations/src/`, 2016.

[33] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[34] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, 2008.

[35] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews. The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database. *PVLDB*, 9(13):1401–1412, 2016.

[36] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, R. N. Sidebotham, et al. The episode file system. In *USENIX Winter 1992 Technical Conference*, pages 43–60, 1992.

[37] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, et al. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, page 8. ACM, 2011.

[38] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper,

S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, et al. The mixed workload CH-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, page 8. ACM, 2011.

[39] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *Acm Sigmod Record*, volume 14, pages 268–279. ACM, 1985.

[40] A. Craig, G. Soules, J. Goodson, and G. Strunk. Metadata efficiency in versioning file systems. In *USENIX Conference on File and Storage Technologies*, 2003.

[41] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984.

[42] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 918–934, New York, NY, USA, 2019. ACM.

[43] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1243–1254, 2013.

[44] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[45] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das. Supporting table partitioning by reference in oracle. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1111–1122. ACM, 2008.

[46] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[47] P. Frühwirt, M. Huber, M. Mulazzani, and E. R. Weippl. Innodb database forensics. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 1028–1036. IEEE, 2010.

[48] G. Gardarin, E. Simon, and L. Verlaine. Querying real time relational data bases. In *ICC (2)*, pages 757–761, 1984.

[49] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[50] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[51] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 355–364, 2010.

[52] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, 1994.

[53] J. A. Hoffer, R. Venkataraman, and H. Topi. *Modern Database Management, 11/E*. Prentice Hall, 2009.

[54] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proc. of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 351–363, 1986.

[55] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: execute-verify replication for multi-core servers. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 237–250, 2012.

[56] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *IEEE International Conference on Data Engineering (ICDE)*, pages 195–206, 2011.

[57] P. Kumar, S. Peri, and K. Vidyasankar. A timestamp based multi-version stm algorithm. In *Proc. International Conference on Distributed Computing and Networking (ICDN)*, pages 212–226, 2014.

[58] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[59] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[60] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, 2014.

[61] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016*, pages 3:1–3:8, 2016.

[62] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 302–313, 2013.

[63] Y. Li and J. M. Patel. Widetable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[64] R. A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems (TODS)*, 2(1):91–104, 1977.

[65] D. Maier, D. Maier, and J. Stein. Indexing in an object-oriented dbms. In *Proceedings on the 1986 international workshop on Object-oriented database systems*, pages 171–182. IEEE Computer Society Press, 1986.

[66] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *ACM European Conference on Computer Systems*, pages 183–196, 2012.

[67] C. Mohan, H. Pirahesh, and R. Lorie. *Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions*, volume 21. 1992.

[68] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. ACM International Conference on Management of Data (SIGMOD), 2015.

[69] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees. *Acta informatica*, 33(6):547–557, 1996.

[70] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[71] M. T. Özsu. A survey of rdf data management systems. *Frontiers of Computer Science*, 10(3):418–432, 2016.

[72] C. H. Papadimitriou and P. C. Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems (TODS)*, 1984.

[73] M. Pezzini, D. Feinberg, N. Rayner, and R. Edjlali. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. `https://www.gartner.com/doc/2657815/`, 2014.

[74] D. R. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *PVLDB*, 5(12):1850–1861, 2012.

[75] D. P. Reed. Naming and synchronization in a decentralized computer system, 1978.

[76] C. Reid, P. A. Bernstein, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *PVLDB*, 4(11), 2011.

[77] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *PVLDB*, 4(11):795–806, 2011.

[78] N. Shamgunov. The memsql in-memory database system. In *IMDM@ VLDB*, 2014.

[79] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *ACM International Conference on Management of Data (SIGMOD)*, pages 731–742, 2012.

[80] B. Sowell, W. Golab, and M. A. Shah. Minuet: A scalable distributed multiversion b-tree. *PVLDB*, 5(9):884–895, 2012.

[81] J. Srinivasan, S. Das, C. Freiwald, E. I. Chong, M. Jagannath, A. Yalamanchi, R. Krishnan, A.-T. Tran, S. DeFazio, and J. Banerjee. Oracle8i index-organized table and its application to new domains. In *VLDB*, pages 285–296, 2000.

[82] Y. Sun and G. E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 159–173, 2019.

[83] Y. Sun, D. Ferizovic, and G. E. Blelloch. PAM: parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[84] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[85] Z. Wang. Index microbench. `https://github.com/wangziqi2016/index-microbench`, 2017.

[86] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488. ACM, 2018.

[87] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[88] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10:781–792, 2017.