Cache Craftiness for Fast Multicore Key-Value Storage

Yandong Mao, Eddie Kohler[†], Robert Morris

MIT CSAIL, [†]Harvard University

Abstract

We present Masstree, a fast key-value database designed for SMP machines. Masstree keeps all data in memory. Its main data structure is a trie-like concatenation of B^+-trees, each of which handles a fixed-length slice of a variable-length key. This structure effectively handles arbitrary-length possibly-binary keys, including keys with long shared prefixes. B^+-tree fanout was chosen to minimize total DRAM delay when descending the tree and prefetching each tree node. Lookups use optimistic concurrency control, a read-copy-update-like technique, and do not write shared data structures; updates lock only affected nodes. Logging and checkpointing provide consistency and durability. Though some of these ideas appear elsewhere, Masstree is the first to combine them. We discuss design variants and their consequences.

On a 16-core machine, with logging enabled and queries arriving over a network, Masstree executes more than six million simple queries per second. This performance is comparable to that of memcached, a non-persistent hash table server, and higher (often much higher) than that of VoltDB, MongoDB, and Redis.

Categories and Subject Descriptors H.2.4 [*Information Systems*]: DATABASE MANAGEMENT – Concurrency

Keywords multicore; in-memory; key-value; persistent

1. Introduction

Storage server performance matters. In many systems that use a single storage server, that server is often the performance bottleneck [1, 18], so improvements directly improve system capacity. Although large deployments typically spread load over multiple storage servers, single-server performance still matters: faster servers may reduce costs, and may also reduce load imbalance caused by partitioning data among servers. Intermediate-sized deployments may be able to avoid the complexity of multiple servers by using sufficiently fast single servers. A common route to high performance is to use different specialized storage systems for different workloads [4].

This paper presents Masstree, a storage system specialized for key-value data in which all data fits in memory, but must persist across server restarts. Within these constraints, Masstree aims to provide a flexible storage model. It supports arbitrary, variable-length keys. It allows *range queries* over those keys: clients can traverse subsets of the database, or the whole database, in sorted order by key. It performs well on workloads with many keys that share long prefixes. (For example, consider Bigtable [12], which stores information about Web pages under permuted URL keys like "edu.harvard.seas.www/news-events". Such keys group together information about a domain's sites, allowing more interesting range queries, but many URLs will have long shared prefixes.) Finally, though efficient with large values, it is also efficient when values are small enough that disk and network throughput don't limit performance. The combination of these properties could free performance-sensitive users to use richer data models than is common for stores like memcached today.

Masstree uses a combination of old and new techniques to achieve high performance [8, 11, 13, 20, 27–29]. It achieves fast concurrent operation using a scheme inspired by OLFIT [11], Bronson *et al.* [9], and read-copy update [28]. Lookups use no locks or interlocked instructions, and thus operate without invalidating shared cache lines and in parallel with most inserts and updates. Updates acquire only local locks on the tree nodes involved, allowing modifications to different parts of the tree to proceed in parallel. Masstree shares a single tree among all cores to avoid load imbalances that can occur in partitioned designs. The tree is a trie-like concatenation of B^+-trees, and provides high performance even for long common key prefixes, an area in which other tree designs have trouble. Query time is dominated by the total DRAM fetch time of successive nodes during tree descent; to reduce this cost, Masstree uses a wide-fanout tree to reduce the tree depth, prefetches nodes from DRAM to overlap fetch latencies, and carefully lays out data in cache lines to reduce the amount of data needed per node. Operations are logged in batches for crash recovery and the tree is periodically checkpointed.

EuroSys'12, April 10–13, 2012, Bern, Switzerland.

We evaluate Masstree on a 16-core machine with simple benchmarks and a version of the Yahoo! Cloud Serving Benchmark (YCSB) [16] modified to use small keys and values. Masstree achieves six to ten million operations per second on parts A–C of the benchmark, more than 30× as fast as VoltDB [5] or MongoDB [2].

The contributions of this paper are as follows. First, an in-memory concurrent tree that supports keys with shared prefixes efficiently. Second, a set of techniques for laying out the data of each tree node, and accessing it, that reduces the time spent waiting for DRAM while descending the tree. Third, a demonstration that a single tree shared among multiple cores can provide higher performance than a partitioned design for some workloads. Fourth, a complete design that addresses all bottlenecks in the way of million-query-per-second performance.

2. Related work

Masstree builds on many previous systems. OLFIT [11] is a B^{link}-tree [27] with optimistic concurrency control. Each update to a node changes the node's version number. Lookups check a node's version number before and after observing its contents, and retry if the version number changes (which indicates that the lookup may have observed an inconsistent state). Masstree uses this idea, but, like Bronson *et al.* [9], it splits the version number into two parts; this, and other improvements, lead to less frequent retries during lookup.

PALM [34] is a lock-free concurrent B^+-tree with twice the throughput of OLFIT. PALM uses SIMD instructions to take advantage of parallelism *within* each core. Lookups for an entire batch of queries are sorted, partitioned across cores, and processed simultaneously, a clever way to optimize cache usage. PALM requires fixed-length keys and its query batching results in higher query latency than OLFIT and Masstree. Many of its techniques are complementary to our work.

Bohannon *et al.* [8] store parts of keys directly in tree nodes, resulting in fewer DRAM fetches than storing keys indirectly. AlphaSort [29] explores several ideas to minimize cache misses by storing partial keys. Masstree uses a trie [20] like data structure to achieve the same goal.

Rao *et al.* [30] propose storing each node's children in contiguous memory to make better use of cache. Fewer node pointers are required, and prefetching is simplified, but some memory is wasted on nonexistent nodes. Cha *et al.* report that a fast B^+-tree outperforms a CSB^+-tree [10]; Masstree improves cache efficiency using more local techniques.

Data-cache stalls are a major bottleneck for database systems, and many techniques have been used to improve caching [14, 15, 21, 31]. Chen *et al.* [13] prefetch tree nodes; Masstree adopts this idea.

H-Store [25, 35] and VoltDB, its commercial version, are in-memory relational databases designed to be orders of magnitude faster than previous systems. VoltDB partitions data among multiple cores to avoid concurrency, and thus avoids data structure locking costs. In contrast, Masstree shares data among all cores to avoid load imbalances that can occur with partitioned data, and achieves good scaling with lock-free lookups and locally locked inserts.

Shore-MT [24] identifies lock contention as a major bottleneck for multicore databases, and improves performance by removing locks incrementally. Masstree provides high concurrency from the start.

Recent key-value stores [2, 3, 12, 17, 26] provide high performance partially by offering a simpler query and data model than relational databases, and partially by partitioning data over a cluster of servers. Masstree adopts the first idea. Its design focuses on multicore performance rather than clustering, though in principle one could operate a cluster of Masstree servers.

3. System interface

Masstree is implemented as a network key-value storage server. Its requests query and change the mapping of keys to values. Values can be further divided into columns, each of which is an uninterpreted byte string.

Masstree supports four operations: $get_c(k)$, $put_c(k,v)$, $remove(k)$, and $getrange_c(k,n)$. The c parameter is an optional list of column numbers that allows clients to get or set subsets of a key's full value. The *getrange* operation, also called "scan," implements a form of range query. It returns up to n key-value pairs, starting with the next key at or after k and proceeding in lexicographic order by key. *Getrange* is not atomic with respect to inserts and updates. A single client message can include many queries.

4. Masstree

Our key data structure is Masstree, a shared-memory, concurrent-access data structure combining aspects of B^+-trees [6] and tries [20]. Masstree offers fast random access and stores keys in sorted order to support range queries. The design was shaped by three challenges. First, Masstree must efficiently support many key distributions, including variable-length binary keys where many keys might have long common prefixes. Second, for high performance and scalability, Masstree must allow fine-grained concurrent access, and its get operations must never dirty shared cache lines by writing shared data structures. Third, Masstree's layout must support prefetching and collocate important information on small numbers of cache lines. The second and third properties together constitute cache craftiness.

4.1 Overview

A Masstree is a trie with fanout 2^{64} where each trie node is a B^+-tree. The trie structure efficiently supports long keys with shared prefixes; the B^+-tree structures efficiently support short keys and fine-grained concurrency, and their medium fanout uses cache lines effectively.

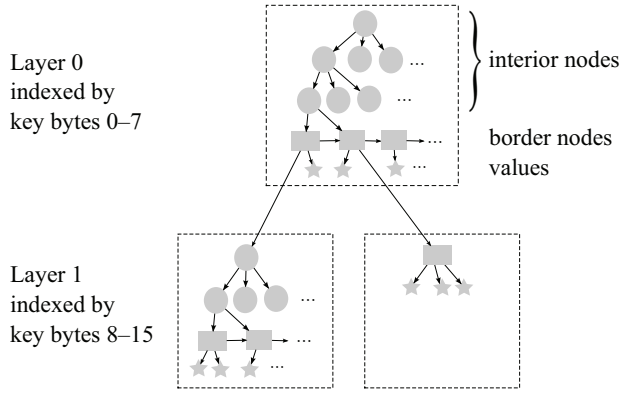


Figure 1. Masstree structure: layers of B$^+$-trees form a trie.

Put another way, a Masstree comprises one or more *layers* of B$^+$-trees, where each layer is indexed by a different 8-byte *slice* of key. Figure 1 shows an example. The trie's single root tree, layer 0, is indexed by the slice comprising key bytes 0–7, and holds all keys up to 8 bytes long. Trees in layer 1, the next deeper layer, are indexed by bytes 8–15; trees in layer 2 by bytes 16–23; and so forth.

Each tree contains at least one *border* node and zero or more *interior* nodes. Border nodes resemble leaf nodes in conventional B$^+$-trees, but where leaf nodes store only keys and values, Masstree border nodes can also store pointers to deeper trie layers.

Keys are generally stored as close to the root as possible, subject to three invariants. (1) Keys shorter than $8h+8$ bytes are stored at layer $\leq h$. (2) Any keys stored in the same layer-h tree have the same 8h-byte prefix. (3) When two keys share a prefix, they are stored at least as deep as the shared prefix. That is, if two keys longer than 8h bytes have the same 8h-byte prefix, then they are stored at layer $\geq h$.

Masstree creates layers as needed (as is usual for tries). Key insertion prefers to use existing trees; new trees are created only when insertion would otherwise violate an invariant. Key removal deletes completely empty trees but does not otherwise rearrange keys. For example, if t begins as an empty Masstree:

1. t.*put*("01234567AB") stores key "01234567AB" in the root layer. The relevant key slice, "01234567", is stored separately from the 2-byte suffix "AB". A *get* for this key first searches for the slice, then compares the suffix.
2. t.*put*("01234567XY"): Since this key shares an 8-byte prefix with an existing key, Masstree must create a new layer. The values for "01234567AB" and "01234567XY" are stored, under slices "AB" and "XY", in a freshly allocated B$^+$-tree border node. This node then replaces the "01234567AB" entry in the root layer. Concurrent gets observe either the old state (with "01234567AB") or the new layer, so the "01234567AB" key remains visible throughout the operation.

```
struct interior_node:
  uint32_t version;
  uint8_t nkeys;
  uint64_t keyslice[15];
  node* child[16];
  interior_node* parent;

union link_or_value:
  node* next_layer;
  [opaque] value;
```

```
struct border_node:
  uint32_t version;
  uint8_t nremoved;
  uint8_t keylen[15];
  uint64_t permutation;
  uint64_t keyslice[15];
  link_or_value lv[15];
  border_node* next;
  border_node* prev;
  interior_node* parent;
  keysuffix_t keysuffixes;
```

Figure 2. Masstree node structures.

3. t.*remove*("01234567XY") traverses through the root layer to the layer-1 B$^+$-tree, where it deletes key "XY". The "AB" key remains in the layer-1 B$^+$-tree.

Balance A Masstree's shape depends on its key distribution. For example, 1000 keys that share a 64-byte prefix generate at least 8 layers; without the prefix they would fit comfortably in one layer. Despite this, Masstrees have the same query complexity as B-trees. Given n keys of maximum length ℓ, query operations on a B-tree examine $O(\log n)$ nodes and make $O(\log n)$ key comparisons; but since each key has length $O(\ell)$, the total comparison cost is $O(\ell \log n)$. A Masstree will make $O(\log n)$ comparisons in each of $O(\ell)$ layers, but each comparison considers *fixed-size* key slices, for the same total cost of $O(\ell \log n)$. When keys have long common prefixes, Masstree outperforms conventional balanced trees, performing $O(\ell + \log n)$ comparisons per query (ℓ for the prefix plus $\log n$ for the suffix). However, Masstree's range queries have higher worst-case complexity than in a B$^+$-tree, since they must traverse multiple layers of tree.

Partial-key B-trees [8] can avoid some key comparisons while preserving true balance. However, unlike these trees, Masstree bounds the number of non-node memory references required to find a key to at most one per lookup. Masstree lookups, which focus on 8-byte key slice comparisons, are also easy to code efficiently. Though Masstree can use more memory on some key distributions, since its nodes are relatively wide, it outperformed our pkB-tree implementation on several benchmarks by 20% or more.

4.2 Layout

Figure 2 defines Masstree's node structures. At heart, Masstree's interior and border nodes are internal and leaf nodes of a B$^+$-tree with width 15. Border nodes are linked to facilitate *remove* and *getrange*. The *version*, *nremoved*, and *permutation* fields are used during concurrent updates and described below; we now briefly mention other features.

The *keyslice* variables store 8-byte key slices as 64-bit integers, byte-swapped if necessary so that native less-than comparisons provide the same results as lexicographic string comparison. This was the most valuable of our coding tricks,

improving performance by 13–19%. Short key slices are padded with 0 bytes.

Border nodes store key slices, lengths, and suffixes. Lengths, which distinguish different keys with the same slice, are a consequence of our decision to allow binary strings as keys. Since null characters are valid within key strings, Masstree must for example distinguish the 8-byte key "ABCDEFG\0" from the 7-byte key "ABCDEFG", which have the same slice representation.

A single tree can store at most 10 keys with the same slice, namely keys with lengths 0 through 8 plus either one key with length > 8 or a link to a deeper trie layer.[1] We ensure that all keys with the same slice are stored in the same border node. This simplifies and slims down interior nodes, which need not contain key lengths, and simplifies the maintenance of other invariants important for concurrent operation, at the cost of some checking when nodes are split. (Masstree is in this sense a restricted type of prefix B-tree [7].)

Border nodes store the suffixes of their keys in *keysuffixes* data structures. These are located either inline or in separate memory blocks; Masstree adaptively decides how much per-node memory to allocate for suffixes and whether to place that memory inline or externally. Compared to a simpler technique (namely, allocating fixed space for up to 15 suffixes per node), this approach reduces memory usage by up to 16% for workloads with short keys and improves performance by 3%.

Values are stored in link_or_value unions, which contain either values or pointers to next-layer trees. These cases are distinguished by the *keylen* field. Users have full control over the bits stored in *value* slots.

Masstree's performance is dominated by the latency of fetching tree nodes from DRAM. Many such fetches are required for a single *put* or *get*. Masstree prefetches all of a tree node's cache lines in parallel before using the node, so the entire node can be used after a single DRAM latency. Up to a point, this allows larger tree nodes to be fetched in the same amount of time as smaller ones; larger nodes have wider fanout and thus reduce tree height. On our hardware, tree nodes of four cache lines (256 bytes, which allows a fanout of 15) provide the highest total performance.

4.3 Nonconcurrent modification

Masstree's tree modification algorithms are based on sequential algorithms for B$^+$-tree modification. We describe them as a starting point.

Inserting a key into a full border node causes a *split*. A new border node is allocated, and the old keys (plus the inserted key) are distributed among the old and new nodes. The new node is then inserted into the old node's parent interior node; if full, this interior node must itself be split (updating its children's *parent* pointers). The split process terminates either at a node with insertion room or at the root, where a new interior node is created and installed. Removing a key simply deletes it from the relevant border node. Empty border nodes are then freed and deleted from their parent interior nodes. This process, like split, continues up the tree as necessary. Though remove in classical B$^+$-trees can redistribute keys among nodes to preserve balance, removal without rebalancing has theoretical and practical advantages [33].

Insert and remove maintain a per-tree doubly linked list among border nodes. This list speeds up range queries in either direction. If only forward range queries were required, a singly linked list could suffice, but the backlinks are required anyway for our implementation of concurrent remove.

We apply common case optimizations. For example, sequential insertions are easy to detect (the item is inserted at the end of a node with no *next* sibling). If a sequential insert requires a split, the old node's keys remain in place and Masstree inserts the new item into an empty node. This improves memory utilization and performance for sequential workloads. (Berkeley DB and others also implement this optimization.)

4.4 Concurrency overview

Masstree achieves high performance on multicore hardware using fine-grained locking and optimistic concurrency control. Fine-grained locking means writer operations in different parts of the tree can execute in parallel: an update requires only local locks.[2] Optimistic concurrency control means reader operations, such as *get*, acquire no locks whatsoever, and in fact *never write to globally-accessible shared memory*. Writes to shared memory can limit performance by causing contention—for example, contention among readers for a node's read lock—or by wasting DRAM bandwidth on writebacks. But since readers don't lock out concurrent writers, readers might observe intermediate states created by writers, such as partially-inserted keys. Masstree readers and writers must cooperate to avoid confusion. The key communication channel between them is a per-node *version* counter that writers mark as "dirty" before creating intermediate states, and then increment when done. Readers snapshot a node's *version* before accessing the node, then compare this snapshot to the *version* afterwards. If the versions differ or are dirty, the reader may have observed an inconsistent intermediate state and must retry.

Our optimistic concurrency control design was inspired by read-copy update [28], and borrows from OLFIT [11] and Bronson *et al.*'s concurrent AVL trees [9].

Masstree's correctness condition can be summarized as *no lost keys*: A *get*(k) operation must return a correct value

[1] At most one key can have length > 8 because of the invariants above: the second such key will create the deeper trie layer. Not all key slices can support 10 keys—any slice whose byte 7 is not null occurs at most twice.

[2] These data structure locks are often called "latches," with the word "lock" reserved for transaction locks. We do not discuss transactions or their locks.

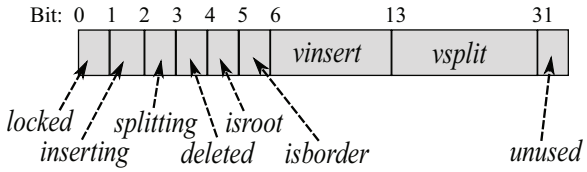


Figure 3. Version number layout. The *locked* bit is claimed by update or insert. *inserting* and *splitting* are "dirty" bits set during inserts and splits, respectively. *vinsert* and *vsplit* are counters incremented after each insert or split. *isroot* tells whether the node is the root of some B^+-tree. *isborder* tells whether the node is interior or border. *unused* allows more efficient operations on the version number.

for k, regardless of concurrent writers. (When *get*(k) and *put*(k,v) run concurrently, the *get* can return either the old or the new value.) The biggest challenge in preserving correctness is concurrent splits and removes, which can shift responsibility for a key away from a subtree even as a reader traverses that subtree.

4.5 Writer–writer coordination

Masstree writers coordinate using per-node spinlocks. A node's lock is stored in a single bit in its *version* counter. (Figure 3 shows the version counter's layout.)

Any modification to a node's keys or values requires holding the node's lock. Some data is protected by other nodes' locks, however. A node's *parent* pointer is protected by its parent's lock, and a border node's *prev* pointer is protected by its previous sibling's lock. This minimizes the simultaneous locks required by split operations; when an interior node splits, for example, it can assign its children's *parent* pointers without obtaining their locks.

Splits and node deletions require a writer to hold several locks simultaneously. When node n splits, for example, the writer must simultaneously hold n's lock, n's new sibling's lock, and n's parent's lock. (The simultaneous locking prevents a concurrent split from moving n, and therefore its sibling, to a different parent before the new sibling is inserted.) As with B^{link}-trees [27], lock ordering prevents deadlock: locks are always acquired up the tree.

We evaluated several writer–writer coordination protocols on different tree variants, including lock-free algorithms relying on compare-and-swap operations. The current locking protocol performs as well or better. On current cache-coherent shared-memory multicore machines, the major cost of locking, namely the cache coherence protocol, is also incurred by lock-free operations like compare-and-swap, and Masstree never holds a lock for very long.

4.6 Writer–reader coordination

We now turn to writer–reader coordination, which uses optimistic concurrency control. Note that even an all-put workload involves some writer–reader coordination, since the initial *put* phase that reaches the node responsible for a key is logically a reader and takes no locks.

```
stableversion(node n):
    v ← n.version
    while v.inserting or v.splitting:
        v ← n.version
    return v

lock(node n):
    while n ≠ NIL and swap(n.version.locked, 1) = 1:
        // retry

unlock(node n):                    // implemented with one memory write
    if n.version.inserting:
        ++n.version.vinsert
    else if n.version.splitting:
        ++n.version.vsplit
    n.version.{locked, inserting, splitting} ← 0

lockedparent(node n):
retry:  p ← n.parent; lock(p)
    if p ≠ n.parent:               // parent changed underneath us
        unlock(p); goto retry
    return p
```

Figure 4. Helper functions.

It's simple to design a correct, though inefficient, optimistic writer–reader coordination algorithm using *version* fields.

1. Before making any change to a node n, a writer operation must mark n.*version* as "dirty." After making its change, it clears this mark and increments the n.*version* counter.
2. Every reader operation first snapshots *every* node's *version*. It then computes, keeping track of the nodes it examines. After finishing its computation (but before returning the result), it checks whether any examined node's *version* was dirty or has changed from the snapshot; if so, the reader must retry with a fresh snapshot.

Universal before-and-after version checking would clearly ensure that readers detect any concurrent split (assuming version numbers didn't wrap mid-computation[3]). It would equally clearly perform terribly. Efficiency is recovered by eliminating unnecessary version changes, by restricting the version snapshots readers must track, and by limiting the scope over which readers must retry. The rest of this section describes different aspects of coordination by increasing complexity.

4.6.1 Updates

Update operations, which change values associated with existing keys, must prevent concurrent readers from observing intermediate results. This is achieved by atomically updat-

[3] Our current counter could wrap if a reader blocked mid-computation for 2^{22} inserts. A 64-bit version counter would never overflow in practice.

```
split(node n, key k):                          // precondition: n locked
        n' ← new border node
        n.version.splitting ← 1
        n'.version ← n.version                 // n' is initially locked
        split keys among n and n', inserting k
ascend: p ← lockedparent(n)                    // hand-over-hand locking
        if p = NIL:                                        // n was old root
            create a new interior node p with children n, n'
            unlock(n); unlock(n'); return
        else if p is not full:
            p.version.inserting ← 1
            insert n' into p
            unlock(n); unlock(n'); unlock(p); return
        else:
            p.version.splitting ← 1
            unlock(n)
            p' ← new interior node
            p'.version ← p.version
            split keys among p and p', inserting n'
            unlock(n'); n ← p; n' ← p'; goto ascend
```

Figure 5. Split a border node and insert a key.

ing values using aligned write instructions. On modern machines, such writes have atomic effect: any concurrent reader will see either the old value or the new value, not some unholy mixture. Updates therefore don't need to increment the border node's version number, and don't force readers to retry.

However, writers must not delete old values until all concurrent readers are done examining them. We solve this garbage collection problem with read-copy update techniques, namely a form of epoch-based reclamation [19]. All data accessible to readers is freed using similar techniques.

4.6.2 Border inserts

Insertion in a conventional B-tree leaf rearranges keys into sorted order, which creates invalid intermediate states. One solution is forcing readers to retry, but Masstree's border-node *permutation* field makes each insert visible in one atomic step instead. This solves the problem by eliminating invalid intermediate states. The *permutation* field compactly represents the correct key order plus the current number of keys, so writers expose a new sort order *and* a new key with a single aligned write. Readers see either the old order, without the new key, or the new order, with the new key in its proper place. No key rearrangement, and therefore no version increment, is required.

The 64-bit *permutation* is divided into 16 four-bit subfields. The lowest 4 bits, *nkeys*, holds the number of keys in the node (0–15). The remaining bits constitute a fifteen-element array, *keyindex*[15], containing a permutation of the numbers 0 through 15. Elements *keyindex*[0] through *keyindex*[*nkeys* − 1] store the indexes of the border node's live keys, in increasing order by key. The other elements list currently-unused slots. To insert a key, a writer locks

```
findborder(node root, key k):
retry:   n ← root; v ← stableversion(n)
         if v.isroot is false:
             root ← root.parent; goto retry
descend: if n is a border node:
             return ⟨n, v⟩
         n' ← child of n containing k
         v' ← stableversion(n')
         if n.version ⊕ v ≤ "locked":  // hand-over-hand validation
             n ← n'; v ← v'; goto descend
         v'' ← stableversion(n)
         if v''.vsplit ≠ v.vsplit:
             goto retry                          // if split, retry from root
         v ← v''; goto descend                   // otherwise, retry from n
```

Figure 6. Find the border node containing a key.

the node; loads the permutation; rearranges the permutation to shift an unused slot to the correct insertion position and increment *nkeys*; writes the new key and value to the previously-unused slot; and finally writes back the new permutation and unlocks the node. The new key becomes visible to readers only at this last step.

A compiler fence, and on some architectures a machine fence instruction, is required between the writes of the key and value and the write of the permutation. Our implementation includes fences whenever required, such as in *version* checks.

4.6.3 New layers

Masstree creates a new layer when inserting a key k_1 into a border node that contains a conflicting key k_2. It allocates a new empty border node n', inserts k_2's current value into it under the appropriate key slice, and then replaces k_2's value in n with the *next_layer* pointer n'. Finally, it unlocks n and continues the attempt to insert k_1, now using the newly created layer n'.

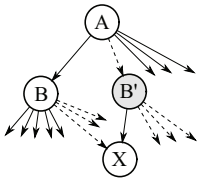
Since this process only affects a single key, there is no need to update n's *version* or *permutation*. However, readers must reliably distinguish true values from *next_layer* pointers. Since the pointer and the layer marker are stored separately, this requires a sequence of writes. First, the writer marks the key as UNSTABLE; readers seeing this marker will retry. It then writes the *next_layer* pointer, and finally marks the key as a LAYER.

4.6.4 Splits

Splits, unlike non-split inserts, remove active keys from a visible node and insert them in another. Without care, a *get* concurrent with the split might mistakenly report these shifting keys as lost. Writers must therefore update *version* fields to signal splits to readers. The challenge is to update these fields in writers, and check them in readers, in such a way that no change is lost.

Figures 5 and 6 present pseudocode for splitting a border node and for traversing down a B$^+$-tree to the border node responsible for a key. (Figure 4 presents some helper functions.) The split code uses hand-over-hand locking and marking [9]: lower levels of the tree are locked and marked as "splitting" (a type of dirty marking) before higher levels. Conversely, the traversal code checks versions hand-over-hand in the opposite direction: higher levels' versions are verified before the traversal shifts to lower levels.

To see why this is correct, consider an interior node B that splits to create a new node B$'$:



(Dashed lines from B indicate child pointers that were shifted to B$'$.) The split procedure changes versions and shifts keys in the following steps.

1. B and B$'$ are marked *splitting*.
2. Children, including X, are shifted from B to B$'$.
3. A (B's parent) is locked and marked *inserting*.
4. The new node, B$'$, is inserted into A.
5. A, B, and B$'$ are unlocked, which increments the A *vinsert* counter and the B and B$'$ *vsplit* counters.

Now consider a concurrent findborder(X) operation that starts at node A. We show that this operation either finds X or eventually retries. First, if findborder(X) traverses to node B$'$, then it will find X, which moved to B$'$ (in step 2) before the pointer to B$'$ was published (in step 4). Instead, assume findborder(X) traverses to B. Since the findborder operation retries on any version difference, and since findborder loads the child's version before double-checking the parent's ("hand-over-hand validation" in Figure 6), we know that findborder loaded B's version before A was marked as *inserting* (step 3). This in turn means that the load of B's version happened before step 1. (That step marks B as *splitting*, which would have caused stableversion to retry.) Then there are two possibilities. If findborder completes before the split operation's step 1, it will clearly locate node X. On the other hand, if findborder is delayed past step 1, it will always detect a split and retry from the root. The B.*version* $\oplus$ *v* check will fail because of B's *splitting* flag; the following stableversion(B) will delay until that flag is cleared, which happens when the split executes step 5; and at that point, B's *vsplit* counter has changed.

Masstree readers treat splits and inserts differently. Inserts retry locally, while splits require retrying from the root. Wide B-tree fanout and fast code mean concurrent splits are rarely observed: in an insert test with 8 threads, less than 1

```
get(node root, key k):
retry:    ⟨n,v⟩ ← findborder(root,k)
forward:  if v.deleted:
              goto retry
          ⟨t,lv⟩ ← extract link_or_value for k in n
          if n.version ⊕ v > "locked":
              v ← stableversion(n); next ← n.next
              while !v.deleted and next ≠ NIL and k ≥ lowkey(next):
                  n ← next; v ← stableversion(n); next ← n.next
              goto forward
          else if t = NOTFOUND:
              return NOTFOUND
          else if t = VALUE:
              return lv.value
          else if t = LAYER:
              root ← lv.next_layer; advance k to next slice
              goto retry
          else: // t = UNSTABLE
              goto forward
```

Figure 7. Find the value for a key.

insert in 10^6 had to retry from the root due to a concurrent split. Other algorithms, such as backing up the tree step by step, were more complex to code but performed no better. However, concurrent inserts are (as one might expect) observed 15× more frequently than splits. It is simple to handle them locally, so Masstree maintains separate split and insert counters to distinguish the cases.

Figure 7 shows full code for Masstree's *get* operation. (Puts are similar, but since they obtain locks, the retry logic is simpler.) Again, the node's contents are extracted between checks of its *version*, and *version* changes cause retries.

Border nodes, unlike interior nodes, can handle splits using their links.[4] The key invariant is that nodes split "to the right": when a border node *n* splits, its *higher* keys are shifted to its new sibling. Specifically, Masstree maintains the following invariants:

- The initial node in a B$^+$-tree is a border node. This node is not deleted until the B$^+$-tree itself is completely empty, and always remains the leftmost node in the tree.
- Every border node *n* is responsible for a range of keys [lowkey(*n*), highkey(*n*)). (The leftmost and rightmost nodes have lowkey(*n*) $= -\infty$ and highkey(*n*) $= \infty$, respectively.) Splits and deletes can modify highkey(*n*), but lowkey(*n*) remains constant over *n*'s lifetime.

Thus, *get* can reliably find the relevant border node by comparing the current key and the next border node's lowkey.

The first lines of findborder (Figure 6) handle stale roots caused by concurrent splits, which can occur at any layer. When the layer-0 global root splits, we update it immediately, but other roots, which are stored in border nodes'

[4] B$^{\text{link}}$-trees [27] and OLFIT [11] also link interior nodes, but our "B$^-$ tree" implementation of remove [33] breaks the invariants that make this possible.

next_layer pointers, are updated lazily during later operations.

4.6.5 Removes

Masstree, unlike some prior work [11, 27], includes a full implementation of concurrent remove. Space constraints preclude a full discussion, but we mention several interesting features.

First, *remove* operations, when combined with inserts, must sometimes cause readers to retry! Consider the following threads running in parallel on a one-node tree:

```
get(n,k1):
  locate k1 at n position i
                                  remove(n,k1):
                                    remove k1 from n position i
  ⋮                               put(n,k2,v2):
                                    insert k2,v2 at n position j
  lv ← n.lv[i]; check n.version;
  return lv.value
```

The *get* operation may return k_1's (removed) value, since the operations overlapped. *Remove* thus must not clear the memory corresponding to the key or its value: it just changes the *permutation*. But then if the *put* operation happened to pick $j = i$, the *get* operation might return v_2, which isn't a valid value for k_1. Masstree must therefore update the *version* counter's *vinsert* field when removed slots are reused.

When a border node becomes empty, Masstree removes it and any resulting empty ancestors. This requires the border-node list be doubly-, not singly-, linked. A naive implementation could break the list under concurrent splits and removes; compare-and-swap operations (some including flag bits) are required for both split and remove, which slightly slows down split. As with any state observable by concurrent readers, removed nodes must not be freed immediately. Instead, we mark them as *deleted* and reclaim them later. Any operation that encounters a *deleted* node retries from the root. *Remove*'s code for manipulating interior nodes resembles that for split; hand-over-hand locking is used to find the right key to remove. Once that key is found, the *deleted* node becomes completely unreferenced and future readers will not encounter it.

Removes can delete entire layer-h trees for $h \geq 1$. These are not cleaned up right away: normal operations lock at most one layer at a time, and removing a full tree requires locking both the empty layer-h tree and the layer-$(h-1)$ border node that points to it. Epoch-based reclamation tasks are scheduled as needed to clean up empty and pathologically-shaped layer-h trees.

4.7 Values

The Masstree system stores values consisting of a version number and an array of variable-length strings called *columns*. Gets can retrieve multiple columns (identified by integer indexes) and puts can modify multiple columns. Multi-column puts are atomic: a concurrent *get* will see either all or none of a *put*'s column modifications.

Masstree includes several value implementations; we evaluate one most appropriate for small values. Each value is allocated as a single memory block. Modifications don't act in place, since this could expose intermediate states to concurrent readers. Instead, put creates a new value object, copying unmodified columns from the old value object as appropriate. This design uses cache effectively for small values, but would cause excessive data copying for large values; for those, Masstree offers a design that stores each column in a separately-allocated block.

4.8 Discussion

More than 30% of the cost of a Masstree lookup is in computation (as opposed to DRAM waits), mostly due to key search within tree nodes. Linear search has higher complexity than binary search, but exhibits better locality. For Masstree, the performance difference of the two search schemes is architecture dependent. On an Intel processor, linear search can be up to 5% faster than binary search. On an AMD processor, both perform the same.

One important PALM optimization is parallel lookup [34]. This effectively overlaps the DRAM fetches for many operations by looking up the keys for a batch of requests in parallel. Our implementation of this technique did not improve performance on our 48-core AMD machine, but on a 24-core Intel machine, throughput rose by up to 34%. We plan to change Masstree's network stack to apply this technique.

5. Networking and persistence

Masstree uses network interfaces that support per-core receive and transmit queues, which reduce contention when short query packets arrive from many clients. To support short connections efficiently, Masstree can configure per-core UDP ports that are each associated with a single core's receive queue. Our benchmarks, however, use long-lived TCP query connections from few clients (or client aggregators), a common operating mode that is equally effective at avoiding network overhead.

Masstree logs updates to persistent storage to achieve persistence and crash recovery. Each server query thread (core) maintains its own log file and in-memory log buffer. A corresponding logging thread, running on the same core as the query thread, writes out the log buffer in the background. Logging thus proceeds in parallel on each core.

A put operation appends to the query thread's log buffer and responds to the client without forcing that buffer to storage. Logging threads batch updates to take advantage of higher bulk sequential throughput, but force logs to storage at least every 200 ms for safety. Different logs may be on different disks or SSDs for higher total log throughput.

Value version numbers and log record timestamps aid the process of log recovery. Sequential updates to a value ob-

tain distinct, and increasing, version numbers. Update version numbers are written into the log along with the operation, and each log record is timestamped. When restoring a database from logs, Masstree sorts logs by timestamp. It first calculates the recovery cutoff point, which is the minimum of the logs' last timestamps, $\tau = \min_{\ell \in L} \max_{u \in \ell} u.timestamp$, where L is the set of available logs and u denotes a single logged update. Masstree plays back the logged updates in parallel, taking care to apply a value's updates in increasing order by version, except that updates with $u.timestamp \geq \tau$ are dropped.

Masstree periodically writes out a checkpoint containing all keys and values. This speeds recovery and allows log space to be reclaimed. Recovery loads the latest valid checkpoint that completed before τ, the log recovery time, and then replays logs starting from the timestamp at which the checkpoint began.

Our checkpoint facility is independent of the Masstree design; we include it to show that persistence need not limit system performance, but do not evaluate it in depth. It takes Masstree 58 seconds to create a checkpoint of 140 million key-value pairs (9.1 GB of data in total), and 38 seconds to recover from that checkpoint. The main bottleneck for both is imbalance in the parallelization among cores. Checkpoints run in parallel with request processing. When run concurrently with a checkpoint, a put-only workload achieves 72% of its ordinary throughput due to disk contention.

6. Tree evaluation

We evaluate Masstree in two parts. In this section, we focus on Masstree's central data structure, the trie of B^+-trees. We show the cumulative impact on performance of various tree design choices and optimizations. We show that Masstree scales effectively and that its single shared tree can outperform separate per-core trees when the workload is skewed. We also quantify the costs of Masstree's flexibility. While variable-length key support comes for free, range query support does not: a near-best-case hash table (which lacks range query support) can provide 2.5× the throughput of Masstree.

The next section evaluates Masstree as a system. There, we describe the performance impact of checkpoint and recovery, and compare the whole Masstree system against other high performance storage systems: MongoDB, VoltDB, Redis, and memcached. Masstree performs very well, achieving 26–1000× the throughput of the other tree-based (range-query-supporting) stores. Redis and memcached are based on hash tables; this gives them $O(1)$ average-case lookup in exchange for not supporting range queries. memcached can exceed Masstree's throughput on uniform workloads; on other workloads, Masstree provides up to 3.7× the throughput of these systems.

6.1 Setup

The experiments use a 48-core server (eight 2.4 GHz six-core AMD Opteron 8431 chips) running Linux 3.1.5. Each core has private 64 KB instruction and data caches and a 512 KB private L2 cache. The six cores in each chip share a 6 MB L3 cache. Cache lines are 64 bytes. Each of the chips has 8 GB of DRAM attached to it. The tests use up to 16 cores on up to three chips, and use DRAM attached to only those three chips; the extra cores are disabled. The goal is to mimic the configuration of a machine more like those easily purchasable today. The machine has four SSDs, each with a measured sequential write speed of 90 to 160 MB/sec. Masstree uses all four SSDs to store logs and checkpoints. The server has a 10 Gb Ethernet card (NIC) connected to a switch. Also on that switch are 25 client machines that send requests over TCP. The server's NIC distributes interrupts over all cores. Results are averaged over three runs.

All experiments in this section use small keys and values. Most keys are no more than 10 bytes long; values are always 1–10 bytes long. Keys are distributed uniformly at random over some range (the range changes by experiment). The key space is not partitioned: a border node generally contains keys created by different clients, and sometimes one client will overwrite a key originally inserted by another. One common key distribution is "1-to-10-byte decimal," which comprises the decimal string representations of random numbers between 0 and 2^{31}. This exercises Masstree's variable-length key support, and 80% of the keys are 9 or 10 bytes long, causing Masstree to create layer-1 trees.

We run separate experiments for gets and puts. Get experiments start with a full store (80–140 million keys) and run for 20 seconds. Put experiments start with an empty store and run for 140 million total puts. Most puts are inserts, but about 10% are updates since multiple clients occasionally put the same key. Puts generally run 30% slower than gets.

6.2 Factor analysis

We analyze Masstree's performance by breaking down the performance gap between a binary tree and Masstree. We evaluate several configurations on 140M-key 1-to-10-byte-decimal get and put workloads with 16 cores. Each server thread generates its own workload: these numbers do not include the overhead of network and logging. Figure 8 shows the results.

Binary We first evaluate a fast, concurrent, lock-free binary tree. Each 40-byte tree node here contains a full key, a value pointer, and two child pointers. The fast jemalloc memory allocator is used.

+Flow, +Superpage, +IntCmp Memory allocation often bottlenecks multicore performance. We switch to Flow, our implementation of the Streamflow [32] allocator ("+Flow"). Flow supports 2 MB x86 superpages, which, when introduced ("+Superpage"), improve throughput by 27–37% due

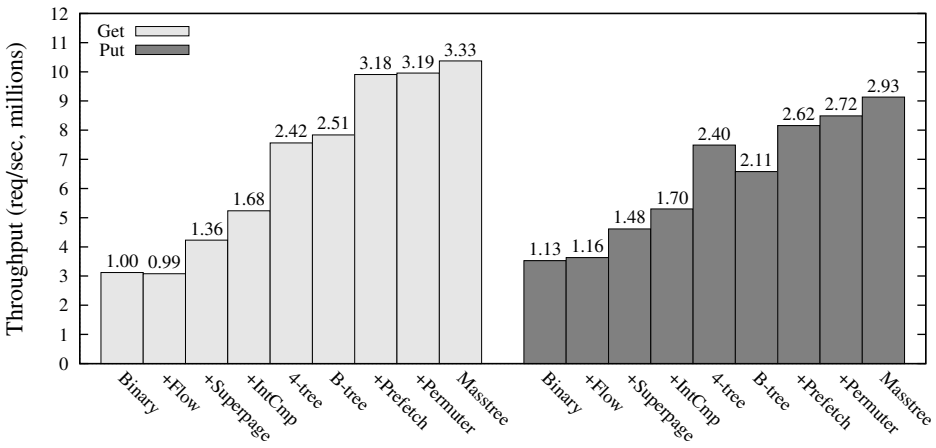


Figure 8. Contributions of design features to Masstree's performance (§6.2). Design features are cumulative. Measurements use 16 cores and each server thread generates its own load (no clients or network traffic). Bar numbers give throughput relative to the binary tree running the get workload.

to fewer TLB misses and lower kernel overhead for allocation. Integer key comparison (§4.2, "+IntCmp") further improves throughput by 15–24%.

4-tree A balanced binary tree has $\log_2 n$ depth, imposing an average of $\log_2 n - 1$ serial DRAM latencies per lookup. We aim to reduce and overlap those latencies and to pack more useful information into cache lines that must be fetched. "4-tree," a tree with fanout 4, uses both these techniques. Its wider fanout nearly halves average depth relative to the binary tree. Each 4-tree node comprises two cache lines, but usually only the first must be fetched from DRAM. This line contains all data important for traversal—the node's four child pointers and the first 8 bytes of each of its keys. (The binary tree also fetches only one cache line per node, but most of it is not useful for traversal.) All internal nodes are full. Reads are lockless and need never retry; inserts are lock-free but use compare-and-swap. "4-tree" improves throughput by 41–44% over "+IntCmp".

B-tree, +Prefetch, +Permuter 4-tree yields good performance, but would be difficult to balance. B-trees have even wider fanout and stay balanced, at the cost of somewhat less efficient memory usage (nodes average 75% full). "B-tree" is a concurrent B^+-tree with fanout 15 that implements our concurrency control scheme from §4. Each node has space for up to the first 16 bytes of each key. Unfortunately this tree *reduces* put throughput by 12% over 4-tree, and does not improve get throughput much. Conventional B-tree inserts must rearrange a node's keys—4-tree never rearranges keys—and B-tree nodes spend 5 cache lines to achieve average fanout 11, a worse cache-line-to-fanout ratio than 4-tree's. However, wide B-tree nodes are easily prefetched to overlap these DRAM latencies. When prefetching is added, B-tree improves throughput by 9–31% over 4-tree ("+Prefetch"). Leaf-node permutations (§4.6.2, "+Permuter") further improve put throughput by 4%.

Masstree Finally, Masstree itself improves throughput by 4–8% over "+Permuter" in these experiments. This surprised us. 1-to-10-byte decimal keys can share an 8-byte prefix, forcing Masstree to create layer-1 trie-nodes, but in these experiments such nodes are quite empty. A 140M-key put workload, for example, creates a tree with 33% of its keys in layer-1 trie-nodes, but the average number of keys per layer-1 trie-node is just 2.3. One might expect this to perform worse than a true B-tree, which has better node utilization. Masstree's design, thanks to features such as storing 8 bytes per key per interior node rather than 16, appears efficient enough to overcome this effect.

6.3 System relevance of tree design

Cache-crafty design matters not just in isolation, but also in the context of a full system. We turn on logging, generate load using network clients, and compare "+IntCmp," the fastest binary tree from the previous section, with Masstree. On 140M-key 1-to-10-byte-decimal workloads with 16 cores, Masstree provides 1.90× and 1.53× the throughput of the binary tree for gets and puts, respectively.[5] Thus, if logging and networking infrastructure are reasonably well implemented, tree design can improve system performance.

6.4 Flexibility

Masstree supports several features that not all key-value applications require, including range queries, variable-length keys, and concurrency. We now evaluate how much these features cost by evaluating tree variants that do not support them. We include network and logging.

[5] Absolute Masstree throughput is 8.03 Mreq/sec for gets (77% of the Figure 8 value) and 5.78 Mreq/sec for puts (63% of the Figure 8 value).

Variable-length keys We compare Masstree with a concurrent B-tree supporting only fixed-size 8-byte keys (a version of "+Permuter"). When run on a 16-core get workload with 80M 8-byte decimal keys, Masstree supports 9.84 Mreq/sec and the fixed-size B-tree 9.93 Mreq/sec, just 0.8% more. The difference is so small likely because the trie-of-trees design effectively has fixed-size keys in most tree nodes.

Keys with common prefixes Masstree is intended to preserve good cache performance when keys share common prefixes. However, unlike some designs, such as partial-key B-trees, Masstree can become superficially unbalanced. Figure 9 provides support for Masstree's choice. The workloads use 16 cores and 80M decimal keys. The X axis gives each test's key length in bytes, but only the final 8 bytes vary uniformly. A 0-to-40-byte prefix is the same for every key. Despite the resulting imbalance, Masstree has 3.4× the throughput of "+Permuter" for relatively long keys. This is because "+Permuter" incurs a cache miss for the suffix of every key it compares. However, Masstree has 1.4× the throughput of "+Permuter" even for 16-byte keys, which "+Permuter" stores entirely inline. Here Masstree's performance comes from avoiding repeated comparisons: it examines the key's first 8 bytes once, rather than $O(\log_2 n)$ times.

Concurrency Masstree uses interlocked instructions, such as compare-and-swap, that would be unnecessary for a single-core store. We implemented a single-core version of Masstree by removing locking, node versions, and interlocked instructions. When evaluated on one core using a 140M-key, 1-to-10-byte-decimal put workload, single-core Masstree beats concurrent Masstree by just 13%.

Range queries Masstree uses a tree to support range queries. If they were not needed, a hash table might be preferable, since hash tables have $O(1)$ lookup cost while a tree has $O(\log n)$. To measure this factor, we implemented a concurrent hash table in the Masstree framework and measured a 16-core, 80M-key workload with 8-byte random alphabetical keys.[6] Our hash table has 2.5× higher total throughput than Masstree. Thus, of these features, only range queries appear inherently expensive.

6.5 Scalability

This section investigates how Masstree's performance scales with the number of cores. Figure 10 shows the results for 16-core get and put workloads using 140M 1-to-10-byte decimal keys. The Y axis shows per-core throughput; ideal scalability would appear as a horizontal line. At 16 cores, Masstree scales to 12.7× and 12.5× its one-core performance for gets and puts respectively.

The limiting factor for the get workload is high and increasing DRAM fetch cost. Each operation consumes about

[6] Digit-only keys caused collisions and we wanted the test to favor the hash table. The hash table is open-coded and allocated using superpages, and has 30% occupancy. Each hash lookup inspects 1.1 entries on average.

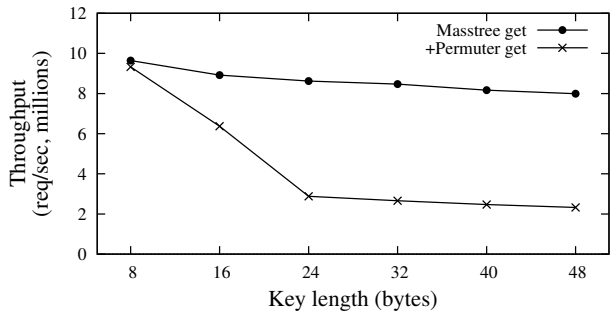


Figure 9. Performance effect of varying key length on Masstree and "+Permuter." For each key length, keys differ only in the last 8 bytes. 16-core get workload.

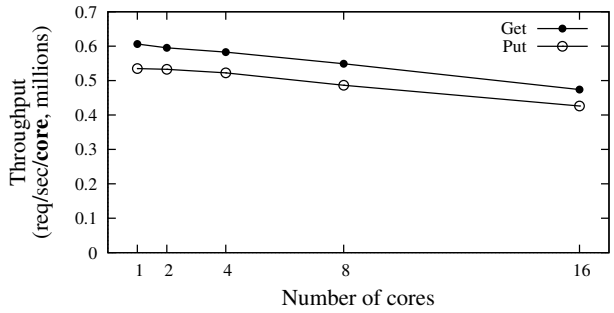


Figure 10. Masstree scalability.

1000 cycles of CPU time in computation independent of the number of cores, but average per-operation DRAM stall time varies from 2050 cycles with one core to 2800 cycles with 16 cores. This increase roughly matches the decrease in performance from one to 16 cores in Figure 10, and is consistent with the cores contending for some limited resource having to do with memory fetches, such as DRAM or interconnect bandwidth.

6.6 Partitioning and skew

Some key-value stores partition data among cores in order to avoid contention. We show here that, while partitioning works well for some workloads, sharing data among all cores works better for others. We compare Masstree with 16 separate instances of the single-core Masstree variant described above, each serving a partition of the overall data. The partitioning is static, and each instance holds the same number of keys. Each instance allocates memory from its local DRAM node. Clients send each query to the instance appropriate for the query's key. We refer this configuration as "hard-partitioned" Masstree.

Tests use 140M-key, 1-to-10-byte decimal get workloads with various partition skewness. Following Hua *et al.* [22], we model skewness with a single parameter δ. For skewness δ, 15 partitions receive the same number of requests, while the last one receives $\delta\times$ more than the others. For example,

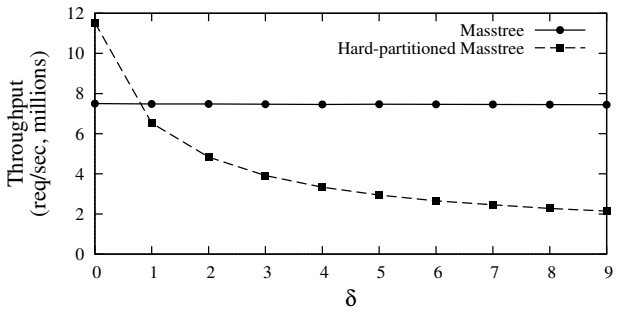


Figure 11. Throughput of Masstree and hard-partitioned Masstree with various skewness (16-core get workload).

at $\delta = 9$, one partition handles 40% of the requests and each other partition handles 4%.

Figure 11 shows that the throughput of hard-partitioned Masstree decreases with skewness. The core serving the hot partition is saturated for $\delta \geq 1$. This throttles the entire system, since other partitions' clients must wait for the slow partition in order to preserve skewness, leaving the other cores partially idle. At $\delta = 9$, 80% of total CPU time is idle. Masstree throughput is constant; at $\delta = 9$ it provides 3.5× the throughput of hard-partitioned. However, for a uniform workload ($\delta = 0$), hard-partitioned Masstree has 1.5× the throughput of Masstree, mostly because it avoids remote DRAM access (and interlocked instructions). Thus Masstree's shared data is an advantage with skewed workloads, but can be slower than hard-partitioning for uniform ones. This problem may diminish on single-chip machines, where all DRAM is local.

7. System evaluation

This section compares the performance of Masstree with that of MongoDB, VoltDB, memcached, and Redis, all systems that have reputations for high performance. Many of these systems support features that Masstree does not, some of which may bottleneck their performance. We disable other systems' expensive features when possible. Nevertheless, the comparisons in this section are not entirely fair. We provide them to put Masstree's throughput in the context of other systems used in practice for key-value workloads.

Figure 12 summarizes the software versions we tested. The client libraries vary in their support for batched or pipelined queries, which reduce networking overheads. The memcached client library does not support batched puts.

Except for Masstree, these systems' storage data structures are not designed to scale well when shared among multiple cores. They are intended to be used with multiple instances on multicore machines, each with a partition of the data. For each system, we use the configuration on 16 cores that yields the highest performance: eight MongoDB processes and one configuration server; four VoltDB processes,

Server	C/C++ client library	Batched query	Range query
MongoDB-2.0	2.0	No	Yes
VoltDB-2.0	1.3.6.1	Yes	Yes
memcached-1.4.8	1.0.3	Yes for get	No
Redis-2.4.5	latest hiredis	Yes	No

Figure 12. Versions of tested servers and client libraries.

each with four sites; 16 Redis processes; and 16 memcached processes. Masstree uses 16 threads.

VoltDB is an in-memory RDBMS. It achieves robustness through replication rather than persistent storage. We turn VoltDB's replication off. VoltDB supports transactions and a richer data and query model than Masstree.

MongoDB is a key-value store. It stores data primarily on disk, and supports named columns and auxiliary indices. We set the MongoDB chunk size to 300MB, run it on an in-memory file system to eliminate storage I/O, and use the "_id" column as the key, indexed by a B-tree.

Redis is an in-memory key-value store. Like Masstree, it logs to disk for crash recovery. We give each Redis process a separate log, using all four SSDs, and disable checkpointing and log rewriting (log rewriting degrades throughput by more than 50%). Redis uses a hash table internally and thus does not support range queries. To implement columns, we used Redis's support for reading and writing specific byte ranges of a value.

memcached is an in-memory key-value store usually used for caching non-persistent data. Like Redis, memcached uses a hash table internally and does not support range queries. The memcached client library supports batched gets but not batched puts, which limits its performance on workloads involving many puts.

Our benchmarks run against databases initialized with 20M key-value pairs. We use two distinct sets of workloads. The first set's benchmarks resemble those in the previous section: get and put workloads with uniformly-distributed 1-to-10-byte decimal keys and 8-byte values. These benchmarks are run for 60 seconds. The second set uses workloads based on the YCSB cloud serving benchmark [16]. We use a Zipfian distribution for key popularity and set the number of columns to 10 and size of each column to 4 bytes. The small column size ensures that no workload is bottlenecked by network or SSD bandwidth. YCSB includes a benchmark, YCSB-E, dependent on range queries. We modify this benchmark to return one column per key, rather than all 10, again to prevent the benchmark from being limited by the network. Initial tests were client limited, so we run multiple client processes. Finally, some systems (Masstree) do not yet support named columns, and on others (Redis) named column support proved expensive; for these systems we modified YCSB to identify columns by number rather than name. We call the result MYCSB.

Workload	Throughput (req/sec, millions, and as % of Masstree)								
	Masstree	MongoDB		VoltDB		Redis		Memcached	
Uniform key popularity, 1-to-10-byte decimal keys, one 8-byte column									
get	9.10	0.04	0.5%	0.22	2.4%	5.97	65.6%	**9.78**	107.4%
put	**5.84**	0.04	0.7%	0.22	3.7%	2.97	50.9%	1.21	20.7%
1-core get	**0.91**	0.01	1.1%	0.02	2.6%	0.54	59.4%	0.77	84.3%
1-core put	**0.60**	0.04	6.8%	0.02	3.6%	0.28	47.2%	0.11	17.7%
Zipfian key popularity, 5-to-24-byte keys, ten 4-byte columns for get, one 4-byte column for update & getrange									
MYCSB-A (50% get, 50% put)	**6.05**	0.05	0.9%	0.20	3.4%	2.13	35.2%	N/A	
MYCSB-B (95% get, 5% put)	**8.90**	0.04	0.5%	0.20	2.3%	2.69	30.2%	N/A	
MYCSB-C (all get)	**9.86**	0.05	0.5%	0.21	2.1%	2.70	27.4%	5.28	53.6%
MYCSB-E (95% getrange, 5% put)	**0.91**	0.00	0.1%	0.00	0.1%	N/A		N/A	

Figure 13. System comparison results. All benchmarks run against a database initialized with 20M key-value pairs and use 16 cores unless otherwise noted. Getrange operations retrieve one column for n adjacent keys, where n is uniformly distributed between 1 and 100.

Puts in this section's benchmarks modify existing keys' values, rather than inserting new keys. This made it easier to preserve MYCSB's key popularity distribution with multiple client processes.

We do not run systems on benchmarks they don't support. The hash table stores can't run MYCSB-E, which requires range queries, and memcached can't run MYCSB-A and -B, which require individual-column update. In all cases Masstree includes logging and network I/O.

Figure 13 shows the results. Masstree outperforms the other systems on almost all workloads, usually by a substantial margin. The exception is that on a get workload with uniformly distributed keys and 16 cores, memcached has 7.4% better throughput than Masstree. This is because memcached, being partitioned, avoids remote DRAM access (see §6.6). When run on a single core, Masstree slightly exceeds the performance of this version of memcached (though as we showed above, a hash table could exceed Masstree's performance by 2.5×).

We believe these numbers fairly represent the systems' absolute performance. For example, VoltDB's performance on uniform key distribution workloads is consistent with that reported by the VoltDB developers for a similar benchmark, volt2 [23].[7]

Several conclusions can be drawn from the data. Masstree has good efficiency even for challenging (non-network-limited) workloads. Batched query support is vital on these benchmarks: memcached's update performance is significantly worse than its get performance, for example. VoltDB's range query support lags behind its support for pure gets. As we would expect given the results in §6.6, partitioned stores perform better on uniform workloads than skewed workloads: compare Redis and memcached on the uniform get workload with the Zipfian MYCSB-C workload.

[7] We also implemented volt2; it gave similar results.

8. Conclusions

Masstree is a persistent in-memory key-value database. Its design pays particular attention to concurrency and to efficiency for short and simple queries. Masstree keeps all data in memory in a tree, with fanout chosen to minimize total DRAM delay when descending the tree with prefetching. The tree is shared among all cores to preserve load balance when key popularities are skewed. It maintains high concurrency using optimistic concurrency control for lookup and local locking for updates. For good performance for keys with long shared prefixes, a Masstree consists of a trie-like concatenation of B^+-trees, each of the latter supporting only fixed-length keys for efficiency. Logging and checkpointing provide consistency and durability.

On a 16-core machine, with logging enabled and queries arriving over a network, Masstree executes more than six million simple queries per second. This performance is comparable to that of memcached, a non-persistent hash table server, and higher (often much higher) than that of VoltDB, MongoDB, and Redis.

Acknowledgments

We thank the Eurosys reviewers and our shepherd, Eric Van Hensbergen, for many helpful comments. This work was partially supported by the National Science Foundation (awards 0834415 and 0915164) and by Quanta Computer. Eddie Kohler's work was partially supported by a Sloan Research Fellowship and a Microsoft Research New Faculty Fellowship.

References

[1] Sharding for startups. `http://www.startuplessonslearned.com/2009/01/sharding-for-startups.html`.

[2] MongoDB. `http://mongodb.com`.

[3] Redis. `http://redis.io`.

[4] Cassandra @ Twitter: An interview with Ryan King. http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king.

[5] VoltDB, the NewSQL database for high velocity applications. http://voltdb.com.

[6] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proc. 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141.

[7] R. Bayer and K. Unterauer. Prefix *B*-Trees. *ACM Transactions on Database Systems*, 2(1):11–26, Mar. 1977.

[8] P. Bohannon, P. McIlroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. *SIGMOD Record*, 30: 163–174, May 2001.

[9] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM PPoPP Symposium*, Bangalore, India, 2010.

[10] S. K. Cha and C. Song. P*TIME: Highly scalable OLTP DBMS for managing update-intensive stream workload. In *Proc. 30th VLDB Conference*, pages 1033–1044, 2004.

[11] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proc. 27th VLDB Conference*, 2001.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26:4:1–4:26, June 2008.

[13] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proc. 2001 SIGMOD Conference*, pages 235–246.

[14] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *Proc. 4th International Workshop on Data Management on New Hardware*, DaMoN '08, pages 25–34, New York, NY, USA, 2008.

[15] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic contention detection and amelioration for data-intensive operations. In *Proc. 2010 SIGMOD Conference*, pages 483–494.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM SOSP*, pages 205–220, 2007.

[18] B. Fitzpatrick. LiveJournal's backend—a history of scaling. http://www.danga.com/words/2005_oscon/oscon-2005.pdf.

[19] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge Computer Laboratory, 2004.

[20] E. Fredkin. Trie memory. *Communications of the ACM*, 3: 490–499, September 1960.

[21] N. Hardavellas, I. Pandis, R. Johnson, N. G. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, Californmia, USA, January 2007.

[22] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proc. 17th VLDB Conference*, pages 525–535, 1991.

[23] J. Hugg. Key-value benchmarking. http://voltdb.com/company/blog/key-value-benchmarking.

[24] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *Proc. 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35, New York, NY, USA, 2009.

[25] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endowment*, 1:1496–1499, August 2008.

[26] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating System Review*, 44:35–40, April 2010.

[27] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[28] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Proc. 2002 Ottawa Linux Symposium*, pages 338–367, 2002.

[29] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A cache-sensitive parallel external sort. *The VLDB Journal*, 4(4):603–627, 1995.

[30] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. *SIGMOD Record*, 29:475–486, May 2000.

[31] K. A. Ross. Optimizing read convoys in main-memory query processing. In *Proc. 6th International Workshop on Data Management on New Hardware*, DaMoN '10, pages 27–33, New York, NY, USA, 2010. ACM.

[32] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proc. 5th International Symposium on Memory Management*, ISMM '06, pages 84–94. ACM, 2006.

[33] S. Sen and R. E. Tarjan. Deletion without rebalancing in balanced binary trees. In *Proc. 21st SODA*, pages 1490–1499, 2010.

[34] 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. *Proc. VLDB Endowment*, 4(11):795–806, August 2011.

[35] M. Stonebraker, S. Madden, J. D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. 33rd VLDB Conference*, pages 1150–1160, 2007.