Scalable and Low Latency Lock-free Data Structures

ALEXANDER KRIZHANOVSKY

Cppcon
The C++ Conference
2022
September 12th-16th

What’s this all about

- Web cache for Tempesta FW (a hybrid of an HTTP accelerator & firewall)
  - `softirq` (*near real-time*)
  - designed for DDoS mitigation (*in-memory*)
  - a lot of data (*persistent*)
- Database data structures assessment
  - lf_hash get performance regression since the bucket size won't decrease
    https://jira.mariadb.org/browse/MDEV-20630
  - PostgreSQL dynamic hash tables (per bucket locks)

Tail latency

e.g. CDN with a 1000 nodes with average request time ~20-50ms
https://tempesta-tech.com/blog/nginx-tail-latency

- 1 / 10,000 requests take more than 2-3 seconds
- 1k nodes with 100KRPS
- **10k users may observe no header image on your site**
- *A sub-second task may take seconds on a busy server with 1 sec secheduling*

Data structure as a database

- Shared cache (each CPU can process a client request to a particular resource)
- Hot path: lookup & insert
- Lookups more than inserts (caching)
- Deletions can be slow, but might block inserts
- Update: delete + insert or per-entity lock + reference count

Lock-free & wait-free

- Lock-free (this talk)
  - guaranteed system-wide progress
  - an operation completes after a finite number of steps
  - waiter helps to finish a conflicting operation
- Wait-free
  - guaranteed system-wide throughput (no starvation)
  - all operations complete after a finite number of steps
  - no livelocking
- Obstruction-free (e.g. transactional memory)
  - abort & retry

Lock-free deletions are tricky

- Intermediary/helping nodes might be concurrently accessed along with the deleted node (insert can construct the whole path and just insert it)
- Concurrent `free()` (e.g. a worker and eviction threads decide to remove the same item)
- Memory fragmentation and/or garbage collection
- Solutions
  - the upper layer responsibility (e.g. by reference counting)
  - RCU
  - Dummy nodes (split-ordered lists, skip trees)

Tempesta DB

- Is part of Tempesta FW (a hybrid of a firewall and web-accelerator)
- Linux kernel space (`softirq` – deferred interrupt context)
- Can be concurrently accessed by many CPUs
- In-memory database
- Simple persistence by dumping `mmap()`’ed areas
  => **offsets** instead of pointers
- Duplicate key entries (stale web responses)
- Multiple indexes (e.g. `URL` or `Vary` for web cache)

Tempesta Technologies

Stored data

- Mostly large string keys **with** or without **ordering requirements**
- Large variable-size records
  - web-cache (`URL` or `Vary` indexes, ordering for `PURGE`)
  - duplicate key entries (stale responses)
- Small fixed-size records
  - client accounts (complicated keys, e.g. `User-Agent` + `IP`)
  - session cookies (short string keys)
  - filter rules (`IP` address)
  - `IP` addresses and network masks

Trees vs hash tables

- **Trie** (tree)
  - ordering
  - range queries
- Hash table
  - fast point queries
  - need rehashing, which is bad for tail latency

Memory allocation: data & index blocks

- There is no need for fast insertion if memory allocation is slow
- Stored entries deletion may lead to memory fragmentation
- Small allocation area, so compress pointers
  Small is beautiful: Techniques to minimise memory footprint - Steven Pigeon - CppCon 2019, https://www.youtube.com/watch?v=Dxy66x6v4HE
- Split index and data blocks
  - spacial locality: sequential accesses within a page
  - data blocks are accessed after index, so keep index blocks together
  - collision traverses data buckets, so keep them together

Binary trees

e.g. `std::map` RB-tree

- Requires rebalancing, rotations involving many nodes
- Hard to implement lock-free
- Hard to implement with fine-grained locking

```
75% lookups, 25% inserts
Hash table with per bucket locks:        80ms avg
std::map with big RW spin-lock:         217ms avg
```

Tempesta Technologies

Hash tables

e.g. `std::unordered_map`

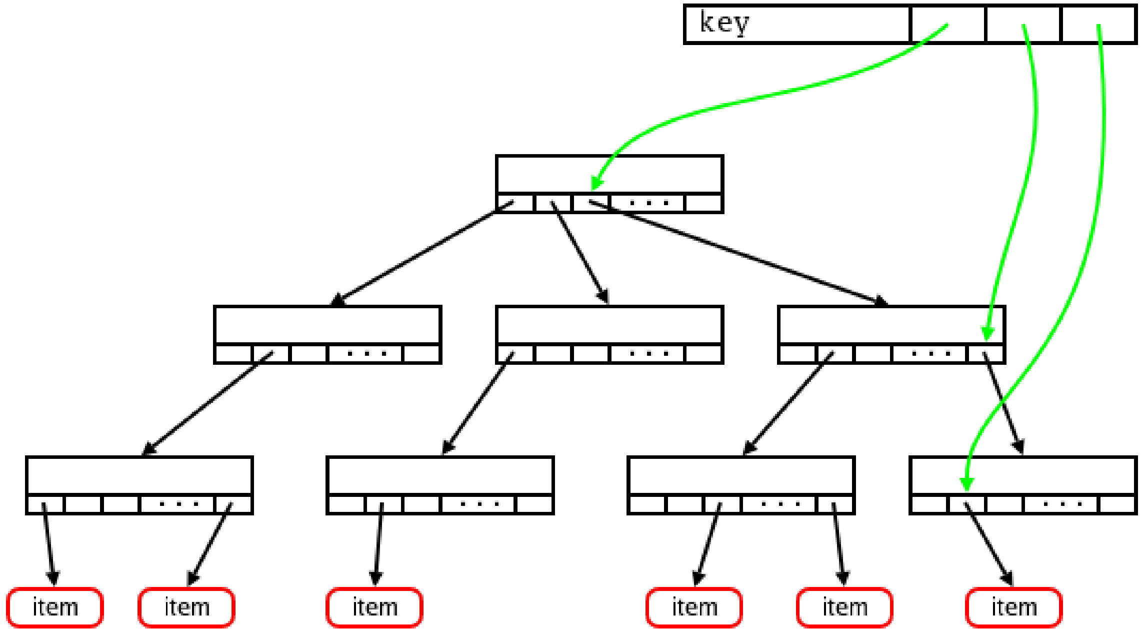
- Bucket chains may grow infinitely
- Rehashing typically takes time and require a global lock
  - great impact to tail latency!
- Easy to implement fine-granular locks (per bucket)

Split-ordered lists

- A lock-free extensible hash table
  - tbb::concurrent_unordered_map
  - MariaDB `rw_trx_hash`
- Uses *persistent* dummy nodes => significant degradation after removal https://jira.mariadb.org/browse/MDEV-20630
- Erasing in tbb::concurrent_unordered_map requires a lock

Radix/patricia tree (trie)

- Judy arrays & ART: 256-way nodes with *adaptive* compression
  "Judy IV Shop Manual", A.Silverstein, 2002
  "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases", V.Leis, 2013
  - *not cache conscious*
  - *hard to make concurrent*



- Height depends on the key length
- No reconstruction (e.g. rebalancing or rehashing)
- **Memory greedy** on
  uniformly distributed keys in a large space
- Easy to make lock-free

Tempesta Technologies

Path compression

- Per-character trie uses to many memory accesses
  `/blog/nginx-tail-latency`
  `/blog/web-cache-poisoning`
- Path compression
  "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases", V.Leis, 2013
- **Burst trie** – no single child nodes

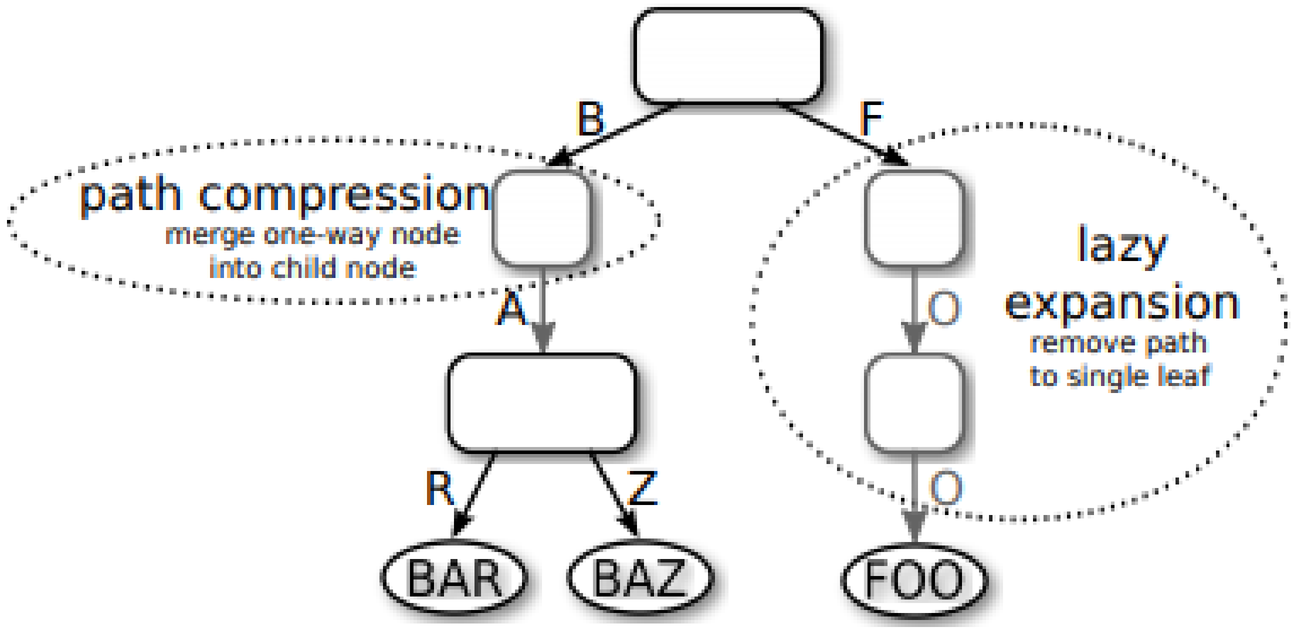


Fig. 6. Illustration of lazy expansion and path compression.

Burst trie

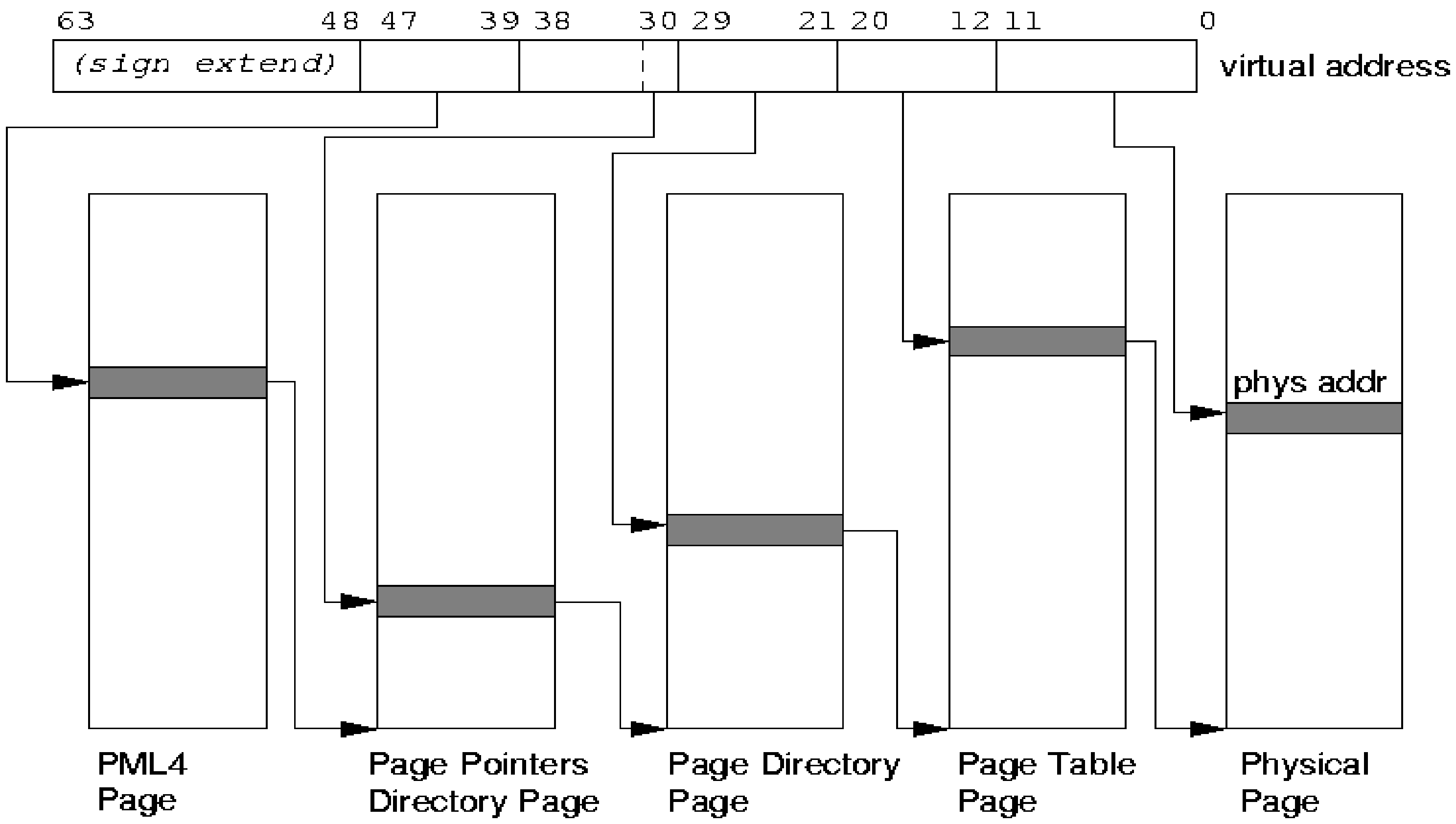
“Burst Tries: A Fast, Efficient Data Structure for String Keys”, S.Heinz, J.Zobel, H.E.Williams, 2002

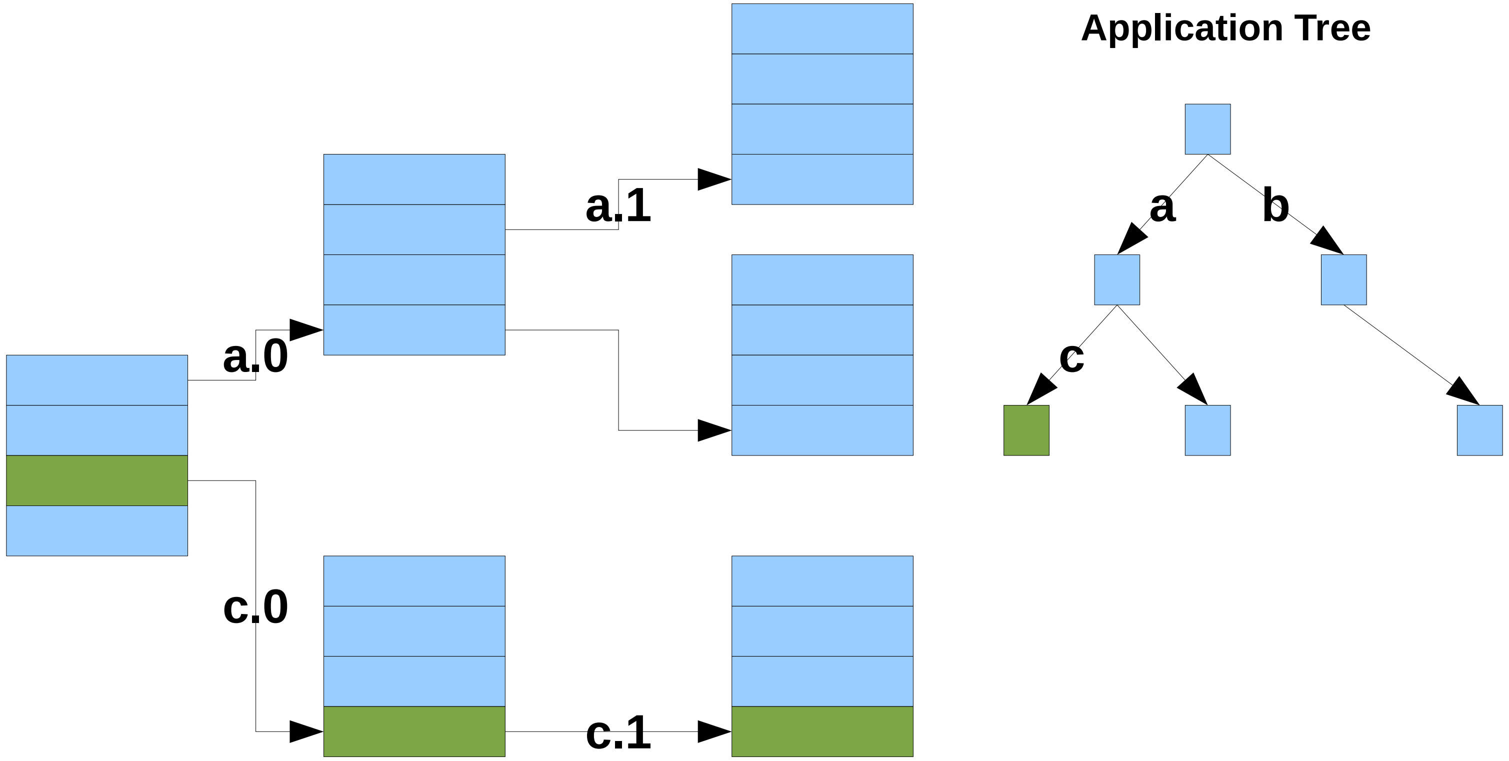
- Collapses trie chains into buckets – better memory usage
  1. use only string prefixes in a trie
  2. resolve collisions by suffixes in a *small* bucket
  3. once the bucket *inefficient* (several heuristics) – *burst* it
- Adaptive: e.g. buckets with rare hits do not burst
- *Poor performance on the start*

X86-64 caches

- Operates with **64 byte** units (cache lines)
- Caches are small and shared
  - associativity (8-way) can make them even smaller
  - e.g. 24 cores/48 threads: L1 - 64KB (per core), L3 - 128MB (shared)
  - access times: L1 ~ 1 cycle, L2 ~ 10 cycles, L3 ~ 50 cycles
  - TLB cache: L1 ~ 1024 pages
- Concurrent update from different CPUs is ~x2 **slower** (`atomics`)
- NUMA remote access is ~x2 slower
- Virtual memory is addressed by **4KB** pages

Tree nodes live inside the radix tree





X86-64 memory ordering

Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide, chapter 8.2

- Neither loads nor stores are reordered with like operations
- Stores are not reordered with earlier loads
- Locked instructions have a total order (`atomics`)
- Loads **may be** reordered with earlier stores to different locations

```
x = y = 0

    CPU1            CPU2

    x  = 1          y  = 1
    r1 = y          r2 = x
```

allowed: `r1 = 0 and r2 = 0`

Tempesta Technologies

Hardware Transaction Memory (Intel TSX)

- Several generations of Intel CPUs, not for AMD
- A transaction may never succeed => **just lock-elision**
- Only for low contended cache lines
- Only for data in L1d cache and if 8-way associativity is enough
- Makes sense for transactions smaller than 32 cache lines
  https://natsys-lab.blogspot.com/2013/11/studying-intel-tsx-performance.html
- (seems) doesn’t work with the modern spin-locks (e.g. MCS locks) (not single integer)?

Tempesta Technologies

Cache conscious data structures

- Node access is access to 1 cache line (L1-L3 data caches)
- Page locality (TLB cache)
- Use a cache line fully on each memory access

Examples of cache conscious data structures

- **CSB$^+$-tree** – cache conscious B$^+$-tree
  ”Making B + -Trees Cache Conscious in Main Memory” by J.Rao and K.A.Ross, 2000
  - pointer to 1st child, all others are by offsets in *contiguous* memory
  - expensive updates
- **FAST** – binary tree with SIMD multi-node comparison
  ”FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs”, C.Kim et al, 2010
- **CST-tree** – cache conscious T-tree
  ”Making T-Trees Cache Conscious on Commodity Microprocessors”, I.Lee, 2011
  - group index and data blocks, use indexes instead of pointers

HAT-trie

“HAT-trie: A Cache-conscious Trie-based Data Structure for Strings”, N.Askitis, R.Sinha, 2007

- Cache-conscious burst trie
  - intermediary nodes
  - buckets as array hashes
    ”Cache-Conscious Collision Resolution in String Hash Tables”, N.Askitis, J.Zobel, 2005
- Hash Array Mapped Tree (HAMT)
  ”Ideal Hash Trees”, P.Bagwell, 2000
- **Can preserve order** (by the cost of constant size)

Keys tradeoffs

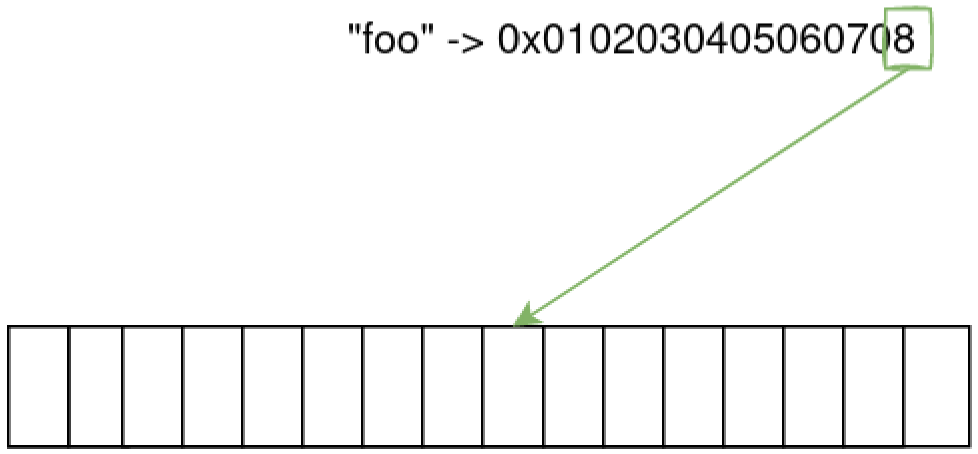
- Fixed-size hash values **vs** ordering (collision: key_{i+1} != key_i+1)
- Constant height (access time) **vs** infinite key length
- Keys distribution is unknown, so perfect hashing is imposible

Memory optimized

- Cache conscious Burst Hash Trie
  - short offsets instead of pointers
  - (almost) lock-free
- lock-free block allocator for virtually contiguous memory

Burst Hash Trie: root

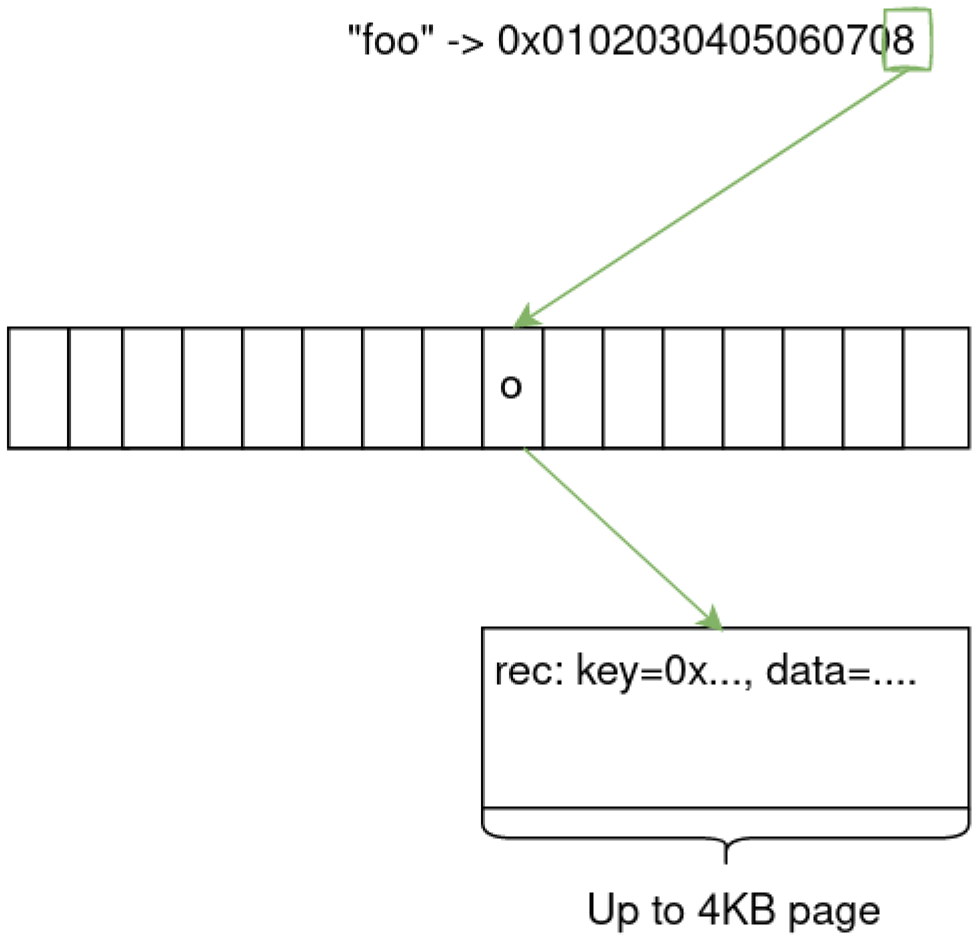
```
const size_t HTRIE_BITS = 4;
const size_t HTRIE_FANOUT = 1 << HTRIE_BITS;

// 2 ^ 31 * 64bytes = 128GB
const size_t HTRIE_DBIT = 1 << (sizeof(int) * 8 - 1);

struct HtrieNode {

    // 16 * 4 = 64 = 1 cache line on x86-64
    unsigned int shifts[HTRIE_FANOUT];

};
```



Tempesta Technologies

Burst HTrie: bucket creation

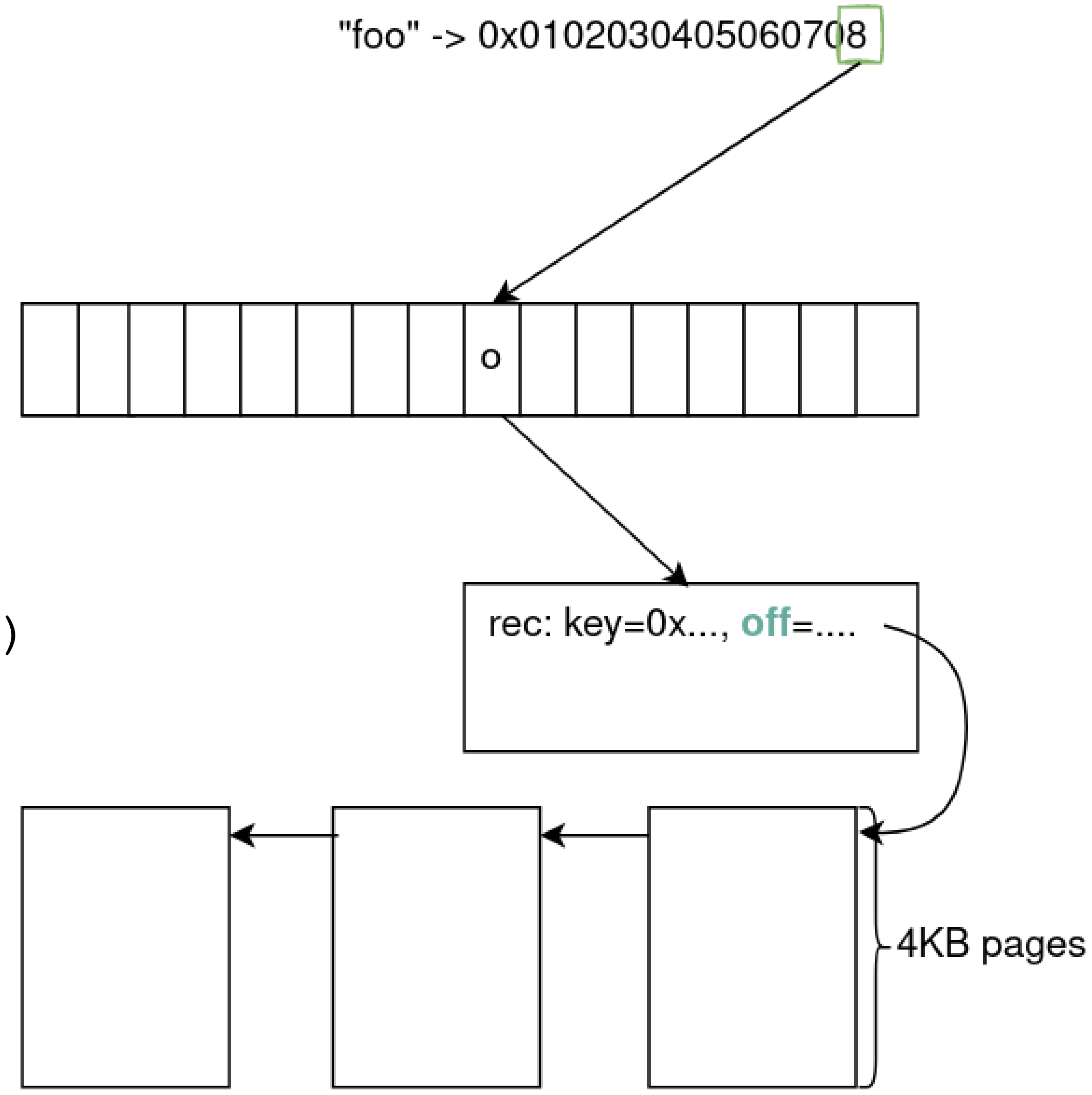
```
struct HtrieBucket {
    unsigned long   col_map; // up to 63 collisions
    unsigned int    next;
};

while (1) {
    node = htrie_descend(key);

    // Alloc and initialize the inserted bucket
    b = htrie_alloc_bucket();
    htrie_bckt_write_data(b, key, data, len, 0, rec);

    // Publish the new node
    unsigned int b_off = add2off(b);
    if (node->shifts[i].compare_exchange(0, b_off) == 0)
        return 0;

    // Somebody already inserted a bucket, rollback
    htrie_rollback_bucket(b);
}
```



Bucket with large/non-inplace data

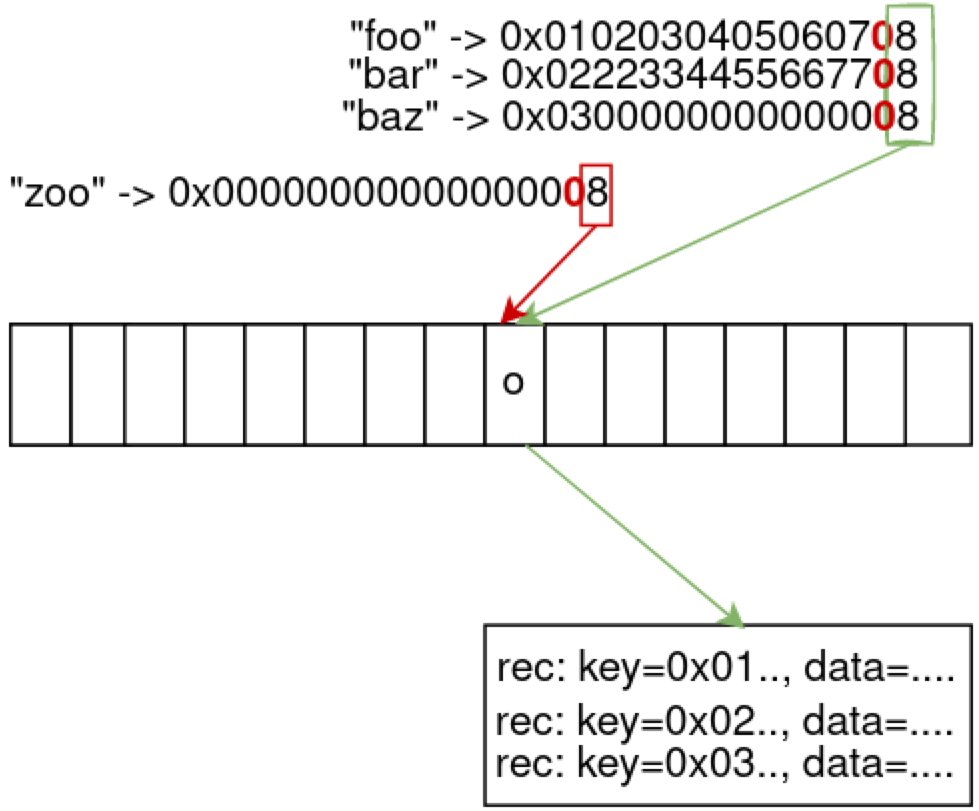
(pointer stability in bursting buckets)

```
// No need to reallocate on the index
// insertion failure
o = htrie_alloc_data(dbh, len)));

rec = htrie_create_rec(dbh, o, key, data, &len);


while (1) {
    node = htrie_descend(key);

    // Alloc and initialize the inserted bucket
    b = htrie_alloc_bucket();
    htrie_bckt_write_data(b, key, data, len, 0, rec)

    // Publish the new node
    unsigned int b_off = add2off(b);
    if (node->shifts[i].compare_exchange(0, b_off)
            == 0)
        return 0;

    // Somebody already inserted a bucket, rollback
    htrie_rollback_bucket(b);

}
```



Bucket burst

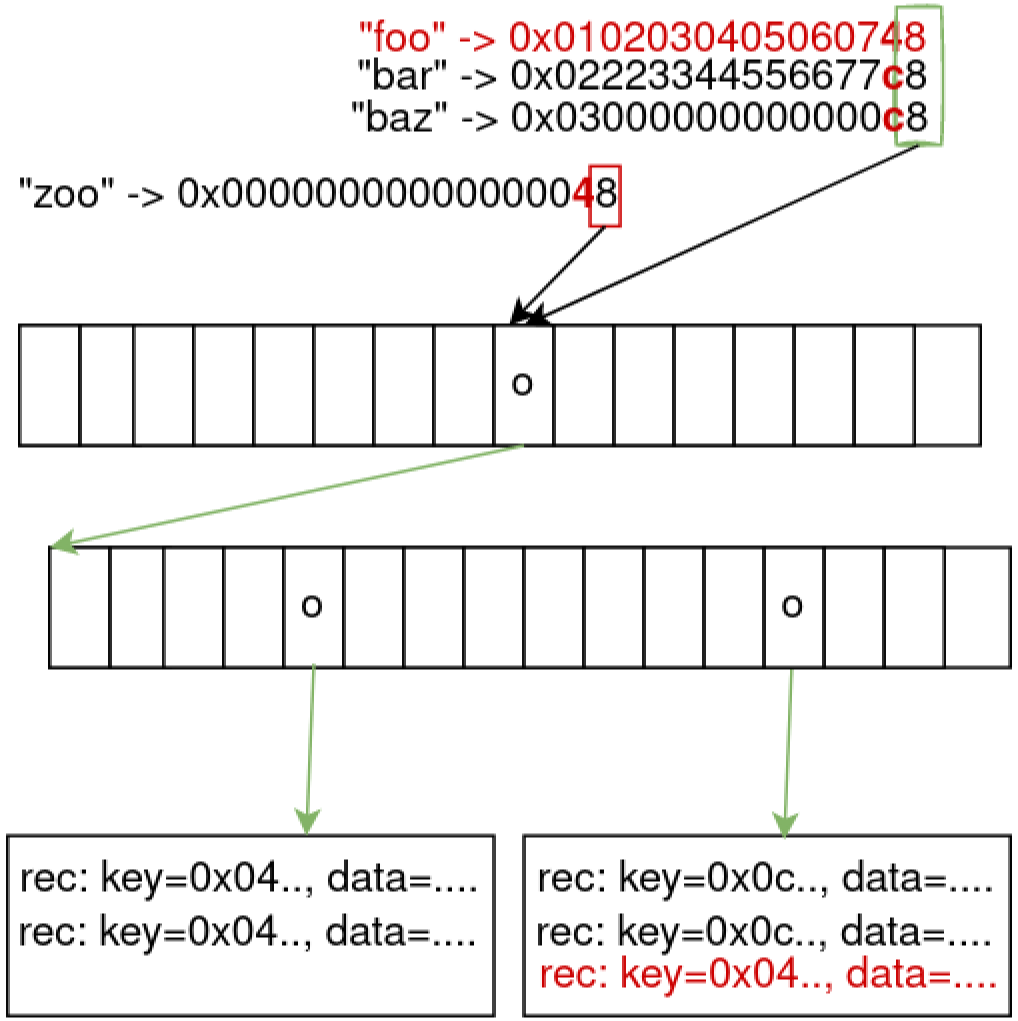
```
retry: // ...descend and all the insertion code...

while (1) {

    // Allocate a new index and bucket nodes,
    // copy buckets data and link from the new index

    // Link the new index with the new & old buckets
    if (CAS(node->shifts[i], new_index))
        goto retry;

    while (1) {
        curr_map = CAS(bucket->col_map, old_map, new_m

        if (curr_map = old_map)
            break;

        map = curr_map ^ map;

        // Copy records for the new collisions

        map = curr_map;

    }
}
```



Tempesta Technologies

Concurrent bucket bursts

```
o = htrie_alloc_index();
in = TDB_PTR(dbh, TDB_II2O(o));
htrie_bckt_move_records(b, map, in, &new_map);

if (node->shifts[i].compare_exchange(old_o, o)
            != old_o)
        goto retry;

// Now old bucket and the new one have the same data

// Other thread can burst old bucket & insert data

While (1) {
    curr_map = b->col_map.compare_exchange(map,
                                           new_map);
    if (curr_map == map) break;

    // Copy records, which we didn't see
    map = curr_map ^ map;

    // Takes care about atomic collision maps
    htrie_bckt_move_records(b, map, in, &new_map);
    map = curr_map;
}
```

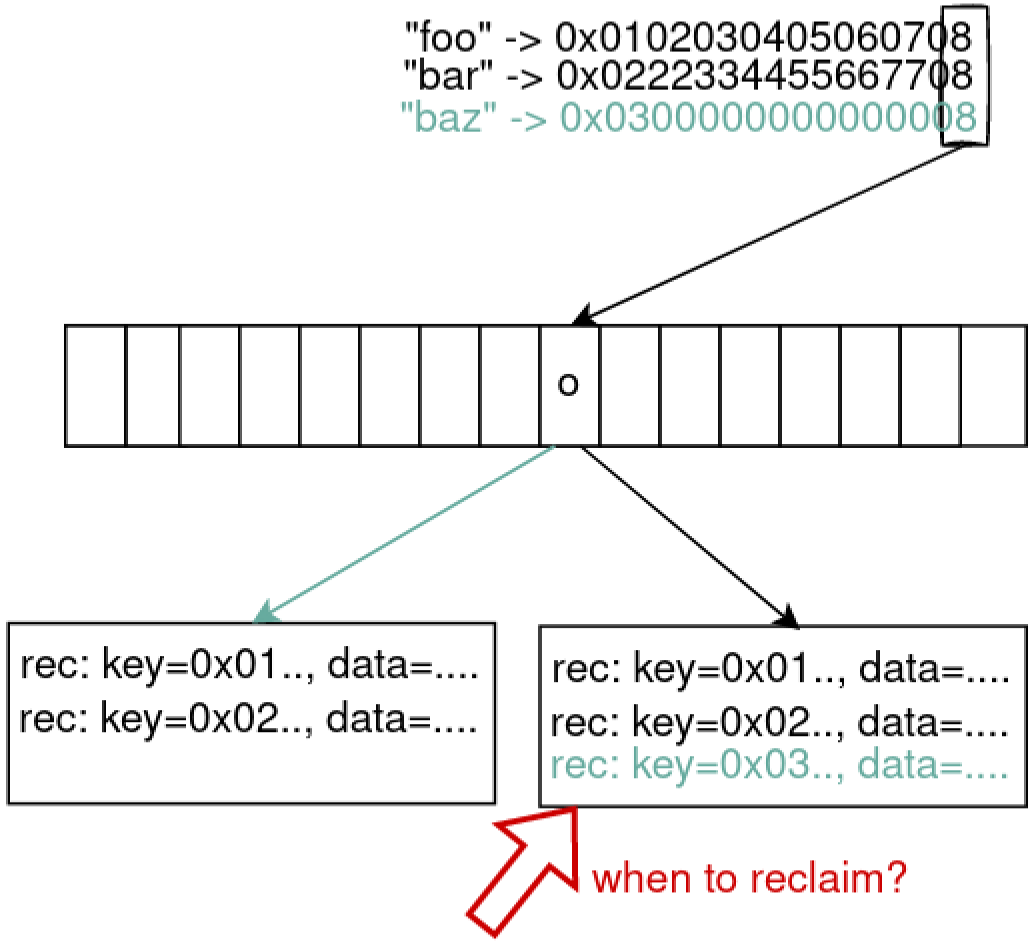


Tempesta Technologies

Reclaiming data

(Don’t delete empty index nodes - just 64B)

```
thread_local local_gen = 0;
atomic<long> global_gen;

htrie_insert() {
    local_gen = global_gen.load();
    // do the stuff
    local_gen = LONG_MAX;
}

htrie_remove() {
    // copy backet and CAS() the index

    bool sync = true;
    do {
        for_each_thread([](){
            if (local_gen >= global_gen.load()) {
                sync = false;

        });
    } while (!sync);

    // reclaim memory
}
```



Tempesta Technologies

Storing data in place vs metadata

- Bursting a bucket storing data
  - large copies
  - a bucket can’t handle several big objects
  - objects change their addresses
- Metadata
  - additional indirection layer (memory access)
  - buckets can be smaller
  - efficient copies

Extensions

- High contention on the root node:
  - we can use 8, 12 or more bits for the root
- Radix tree is the same as patricia tree
  - we can use the trie without the hash function
  - non-constant time access
  - text compression helps to reduce the trie height
- Due to 31-bit offsets, Htrie can address 128GB only
  - Data sharding as tries forest
  - NUMA-aware scheduling

Thank you! Questions?

Availability: *https://github.com/tempesta-tech/blog/tree/master/htrie*

Tempesta FW: *https://github.com/tempesta-tech/tempesta*

Alexander Krizhanovsky

ak@tempesta-tech.com
@a_krizhanovsky

Tempesta Technologies