Scalable and Low Latency Lock-free Data Structures

Alexander Krizhanovsky

Tempesta Technologies, Inc.

ak@tempesta-tech.com

Who am I?

- CEO & founder of **Tempesta Technologies** *(https://tempesta-tech.com/)*
- **Tempesta FW** – ultra-fast & secure Linux kernel web accelerator
  https://github.com/tempesta-tech/tempesta
  - volumetric DDoS protection
  - behavior analysis against web scraping bots
- **Custom high-performance projects**:
  - WAF mentioned in Gartner magic quadrant
  - contributed to MariaDB and Percona XtraDB Cluster engines
  - Ulra-scalable kernel bypass NFS and S3 servers

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

Deterministic data structure is crucial!
(no rehashing)

Tempesta
Technologies

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

Tempesta DB

- 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)

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

Tempesta Technologies

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

Tempesta Technologies

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
  - hazard pointers
  - dummy nodes (split-ordered lists, skip trees)

Tempesta Technologies

Reclaming: hazard pointers

”Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects”,
M.M. Michael, 2004

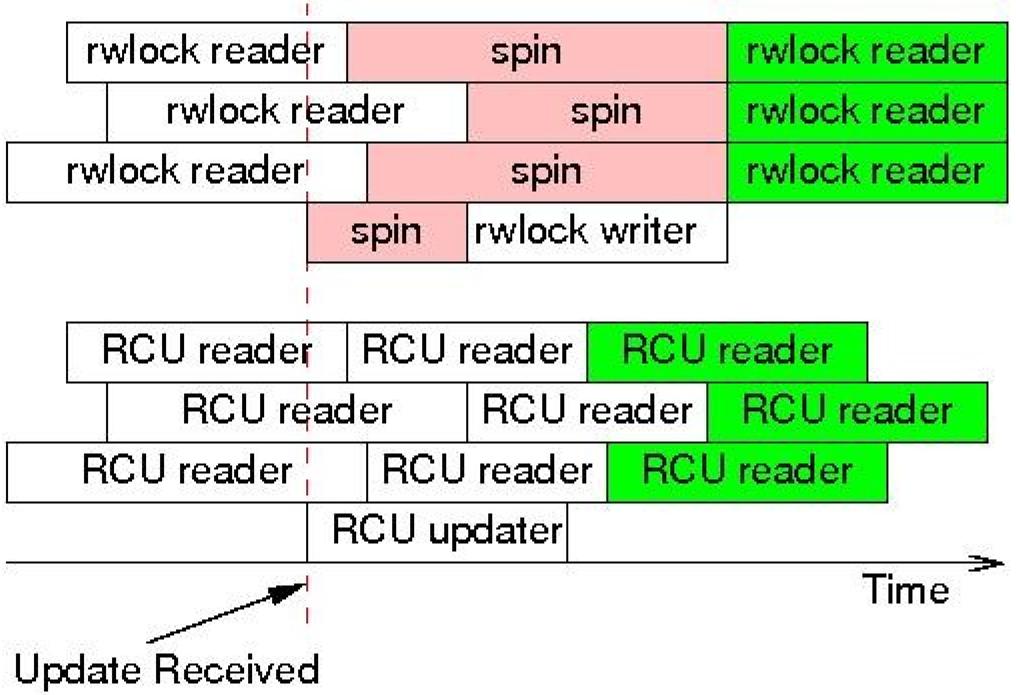
- r/w: second thread-local (*hazard*) pointer
- delete: check all thread-local hazard pointers
- Pros
  - memory can be freed immediately
- Cons (overheads)
  - readers must update hazard pointers
  - every access requires hazard pointer setup
  - requires sophisticated protocol to traverse linked data structures

Reclaming: RCU

(kernel softirq context, `TREE`, `PREEMPT_NONE`)

https://lwn.net/Kernel/Index/#Read-copy-update

- **update**: create a new version of the data and update pointers atomically
- not more than 10% updaters
- Pros
  - no read overhead – almost no-op `rcu_read_lock()`
- Cons
  - reading must be fast
  - defferred feeing may lag



“What is RCU? Part 2: Usage”, P.McKenney, 2007
https://lwn.net/Articles/263130/

Reclaming: `percpu_ref`

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/percpu-refcount.h
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/lib/percpu-refcount.c

- read path: **per-cpu reference counters**
- cleanup: switch to **atomic global counter** to avoid new references
- memory **overhead**: N objects on M CPUs

RCU vs hazard pointers vs `percpu_ref`

"Hazard Pointers for the Linux Kernel?", P.McKenney, https://docs.google.com/document/d/113WFjGlAW4m72xNbZWHUSE-yU2HIJnWpiXp91ShtgeE

"Hazard pointers in Linux kernel", B.Feng, N.Upadhyay, P.McKenney, Linux Plumbers Conference 2024 https://www.youtube.com/watch?v=yoVLSKG2pZs

Property	Reference Counting	Per-CPU Ref Count	RCU	Hazard Pointer
Readers	Slow & unscalable	Fast & scalable	Fast & scalable	Fast & scalable
Memory Overhead	*O(Nobj)*	*O(Nobj*Ncpu)*	*O(Nobj)*	*~O(Nobj)*
Protection Duration	Can be long	Can be long	Bounded duration	Can be long
Traversal Retries	If any object deleted	If any object deleted	Never	If any object deleted
Deferred Memory	None	Switch to global	Can be large	Depends on scan interval

Trees vs hash tables

- **Hash table**
  - fast point queries
  - need rehashing, which is bad for tail latency
- **Tree** (binary tree, B-tree etc)
  - ordering
  - range queries
- **Trie** (patricia/radix **tree**)
  - no need rebalancing

=> effort on faster trie

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
  - ..but we can use trees instead of lists (almost HTrie)
- Rehashing typically takes time and require a global lock
  - great impact to tail latency!
- Easy to implement fine-granular locks (per bucket)
- Open addressing can be SIMD-accelerated
  “Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step”, M.Kulukundis, https://www.youtube.com/watch?v=ncHmEUmJZf4
  - ...but with locking

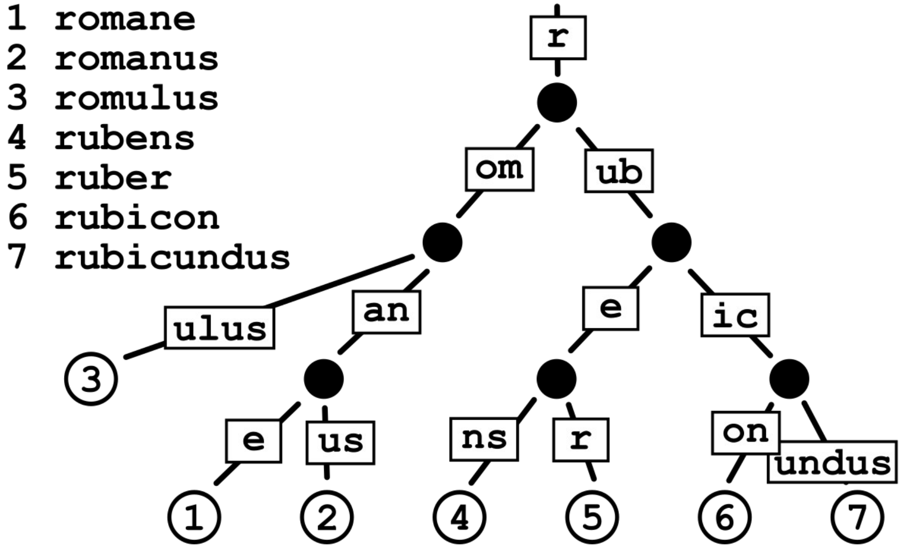
Tempesta Technologies

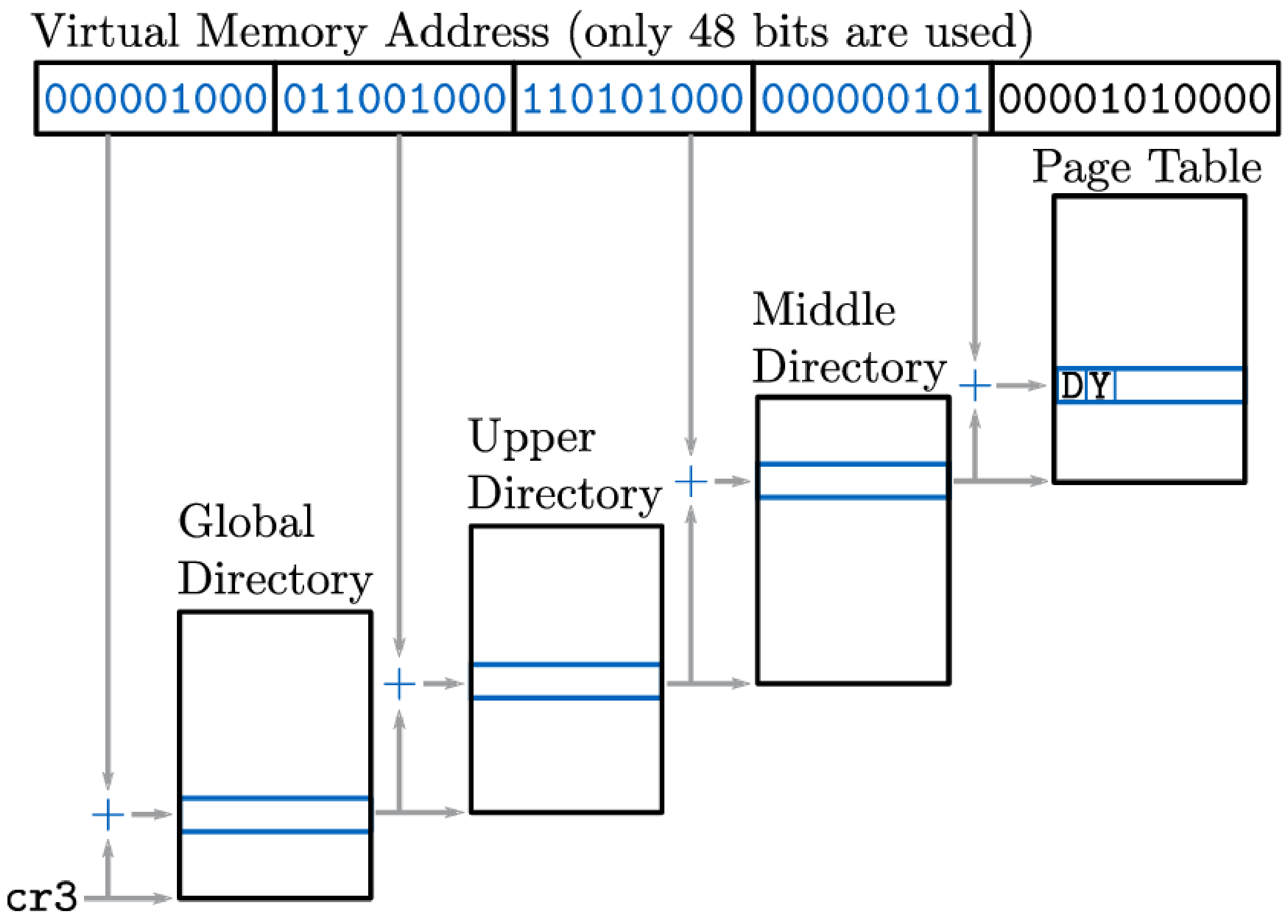
Split-ordered lists

- 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/prefix/patricia tree (trie)

- E.g. page table
- **Memory greedy** on uniformly distributed keys in a large space
  - *Quiz: why* `malloc()`*'ed addresses are close to each other?*
- Height depends on the key length
  - constant search time for integer keys

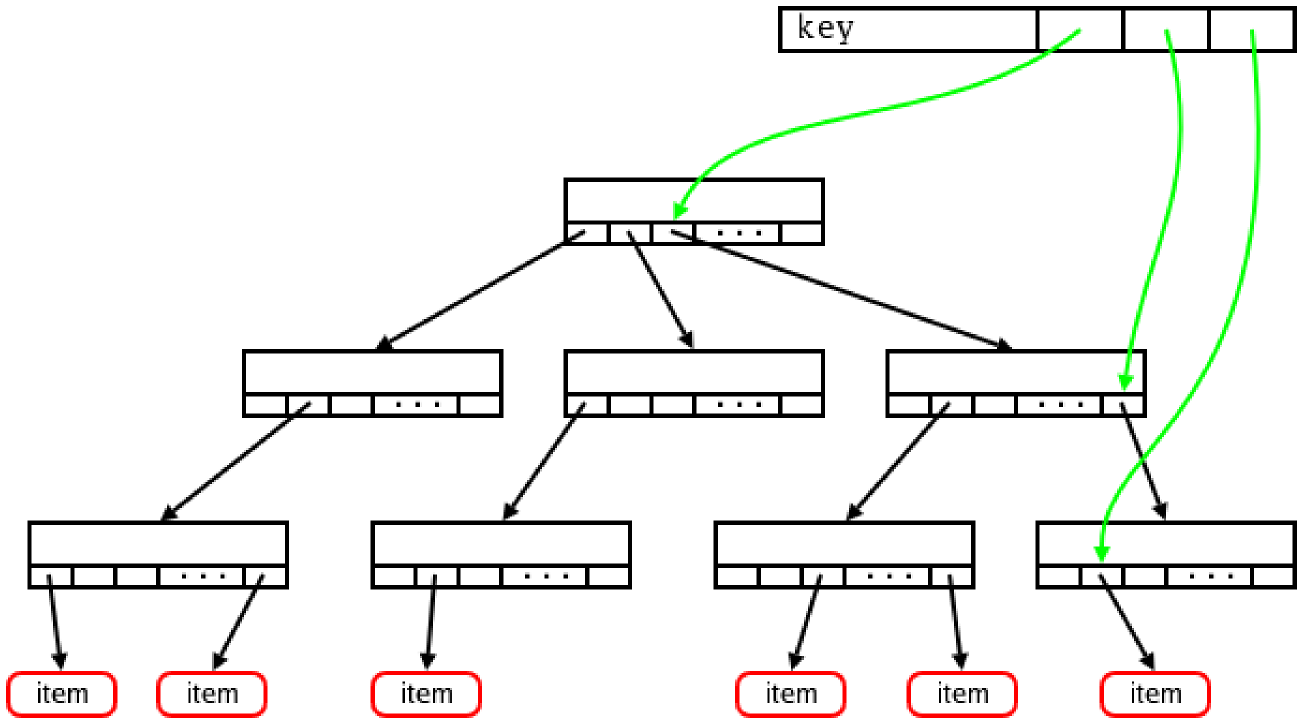




Radix/prefix/patricia tree (trie)

(Tree of hash tables with hash functions as part of the key)

- 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*
- No reconstruction (e.g. rebalancing or rehashing)
- Easy to make lock-free



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

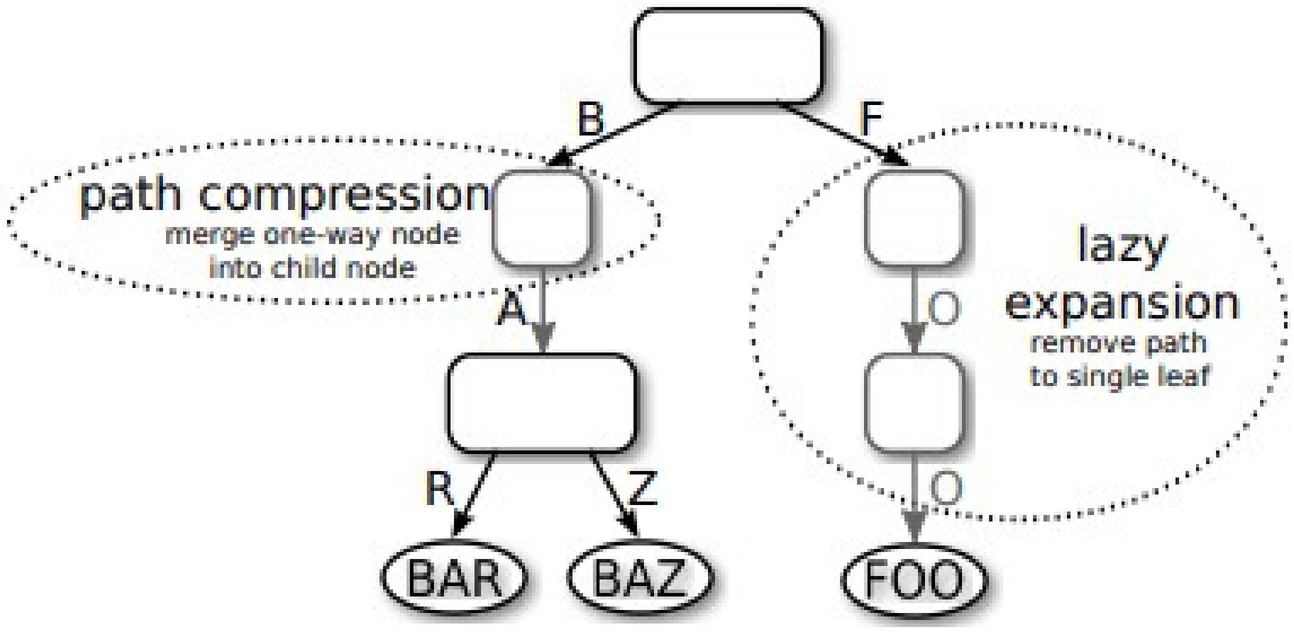
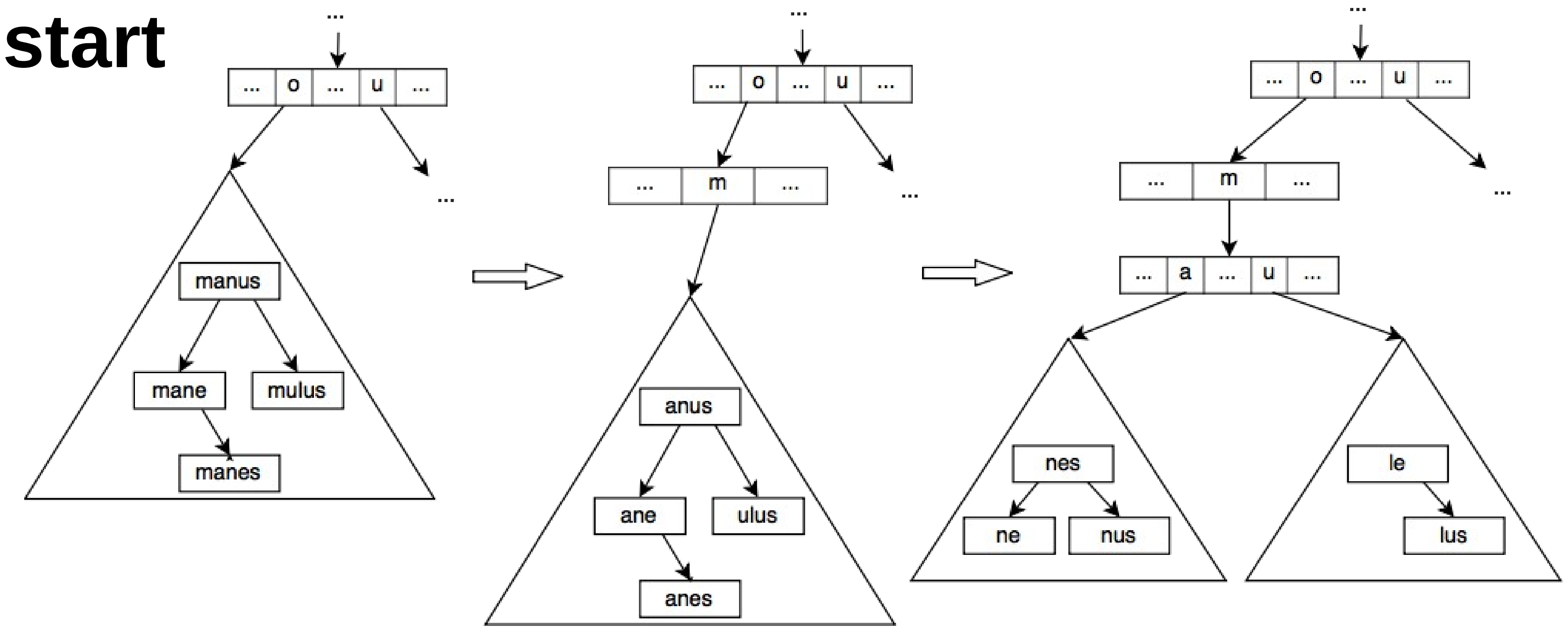


Fig. 6. Illustration of lazy expansion and path compression.

Burst trie

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

- Starts as a small hash table with small fixed collision buckets
- **Burst** a bucket if it’s out of space – create a new hash level
  - Can be adaptive: e.g. burst top-hitting buckets earlier
- **Poor performance on the start**
  => bigger root node
- **Buckets reconstruction**
  (but with good scale)



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)

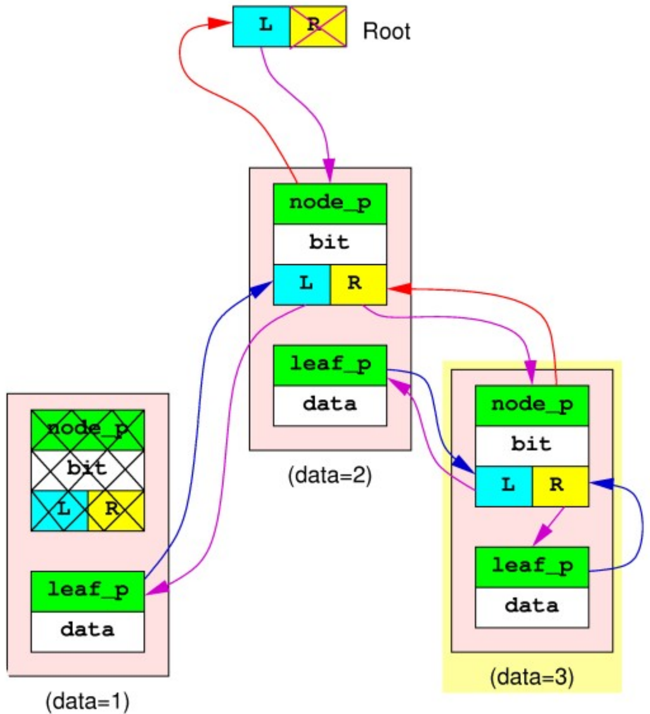
Tempesta Technologies

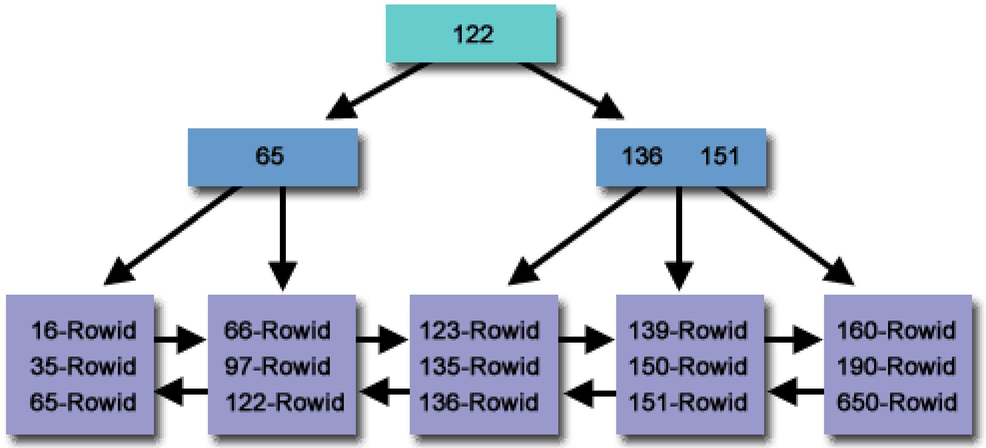
Memory allocation: data & index blocks

- 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

Index & data blocks

- Index with data blocks, e.g. ellastic binary tree
  http://wtarreau.blogspot.com/2011/12/elastic-binary-trees-ebtree.html
  - fewer memory accesses for **small data sets**
- Index in separate blocks, e.g. B-tree
  - faser scans on **large indexes**



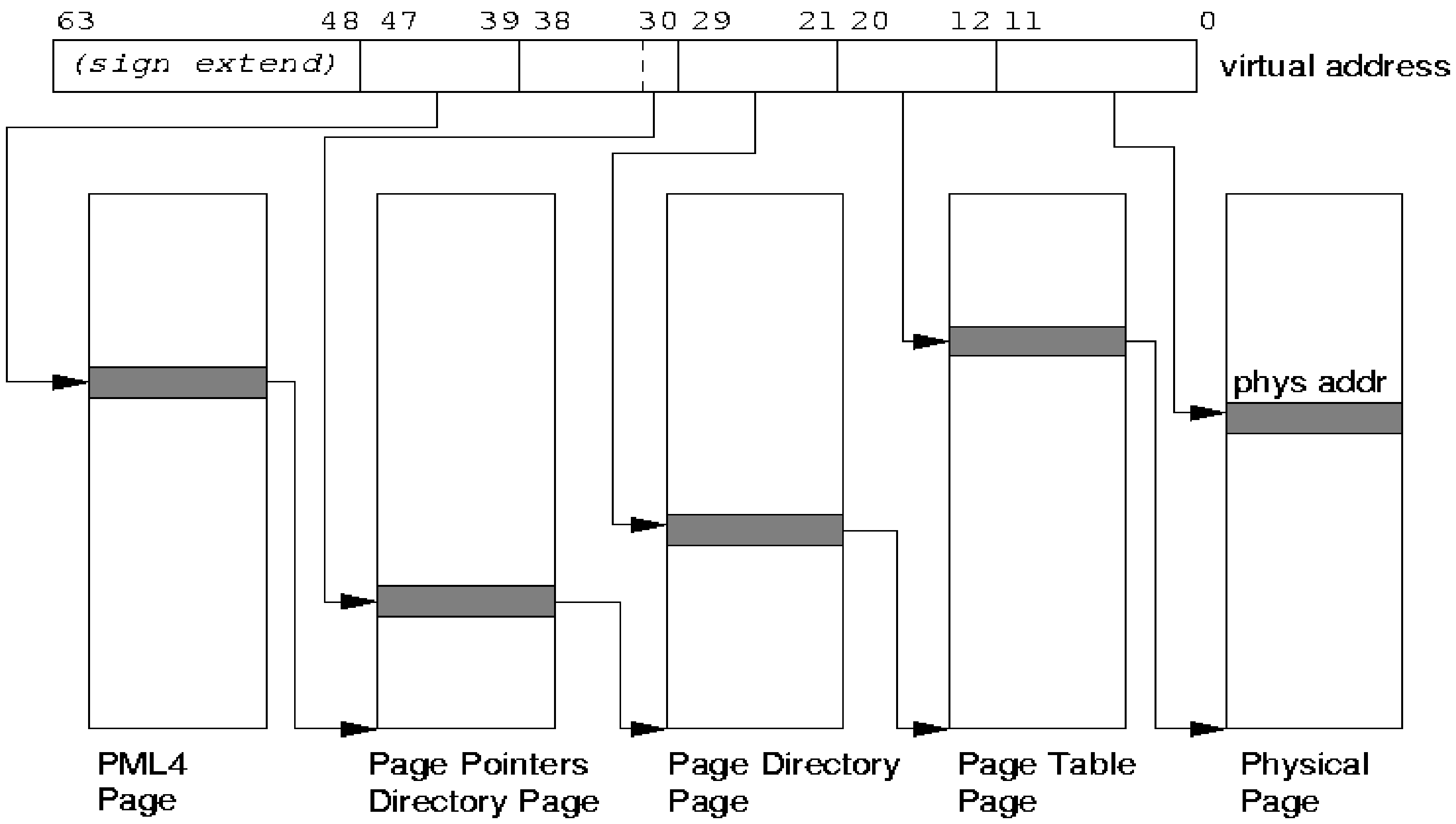


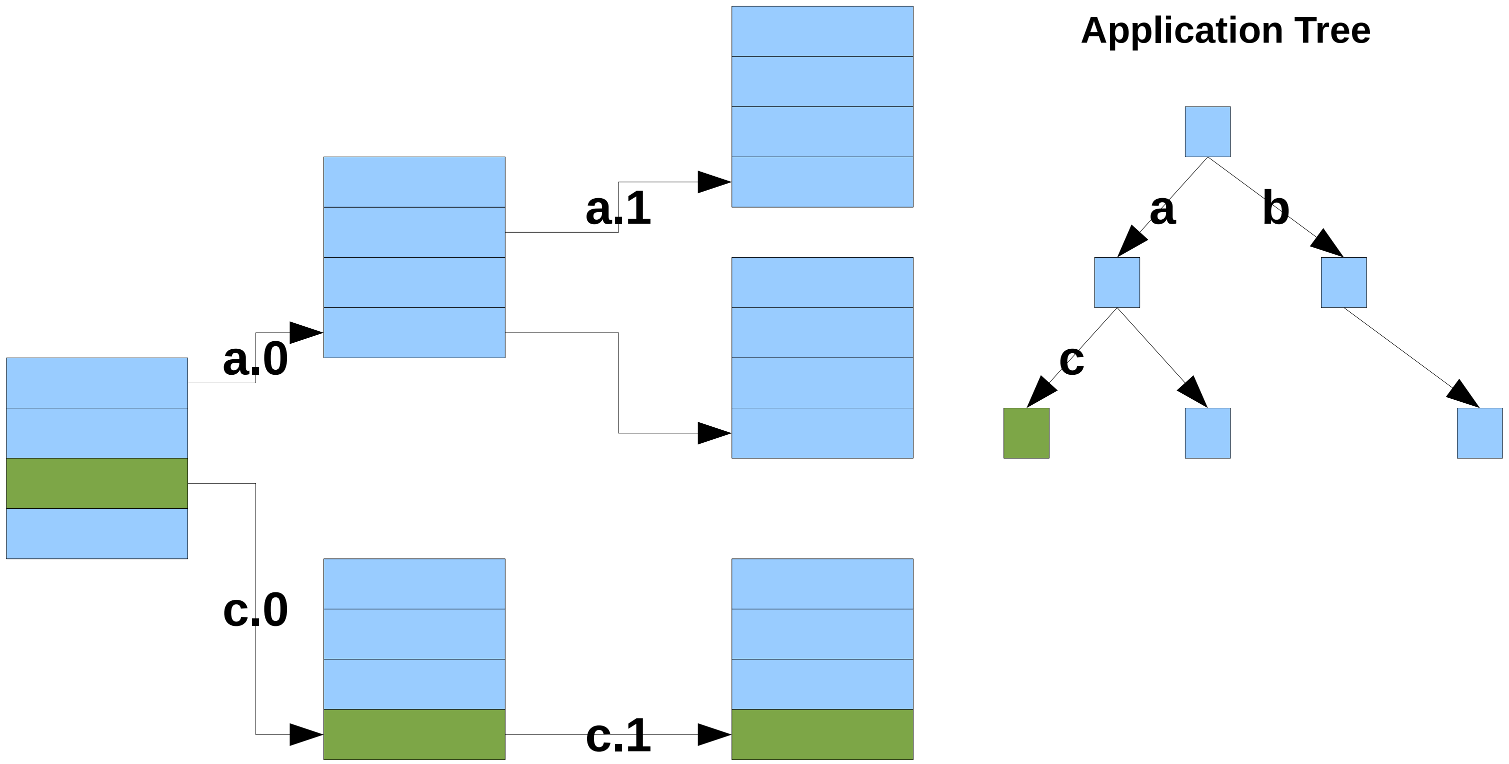
Tempesta Technologies

Being *conscious* about 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





Going lock-free: x86-64 memory ordering

“Abusing Your Memory Model for Fun and Profit”, S.A.Bahra, P.Khuong, CppCon 2019, https://www.youtube.com/watch?v=N07tM7xWF1U

- 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, so only lock-elision
- Only for low contended cache lines
  - doesn’t work with the modern spin-locks
    (e.g. MCS locks) (not single integer)?
- 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
- **Bad and dead since Alder Lake**

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

Cache conscious data structures *(no lock-free)*

- **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

Tempesta Technologies

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**

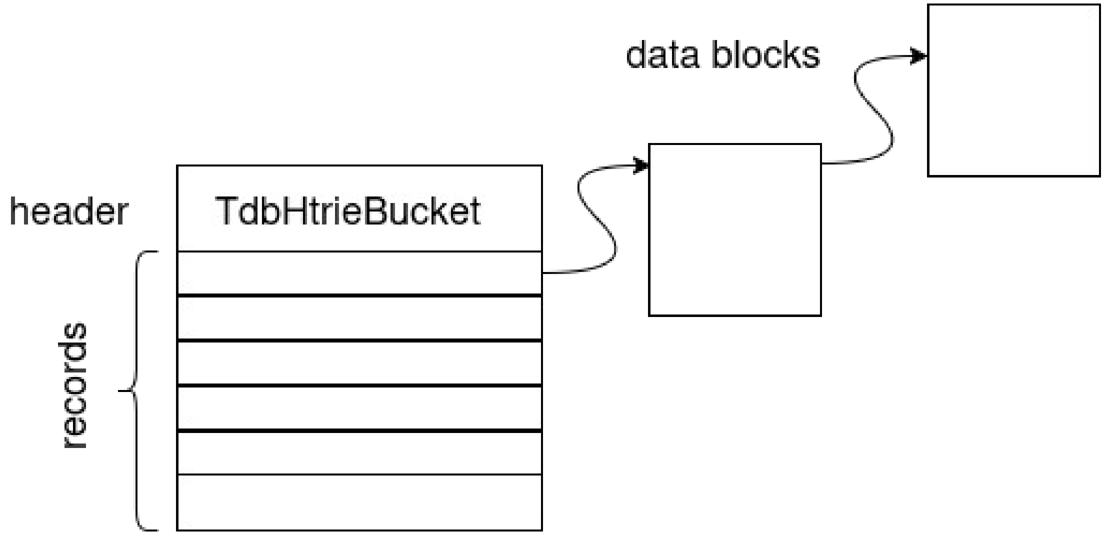
Tempesta DB HTrie

- Cache conscious Burst Hash Trie
- Lock-free
  - lock-free block allocator for virtually contiguous memory
- Persistence: `mmap()`'ed area is dumped eventually
  - short offsets instead of pointers
- Copy on updates
- Hazard(-like) pointers for data reclaiming
- **Pointers stability** vs better CPU cache utilization
  - large stored data is used by pointers
  - small data (e.g. an IP address) can be copied

Tempesta Technologies

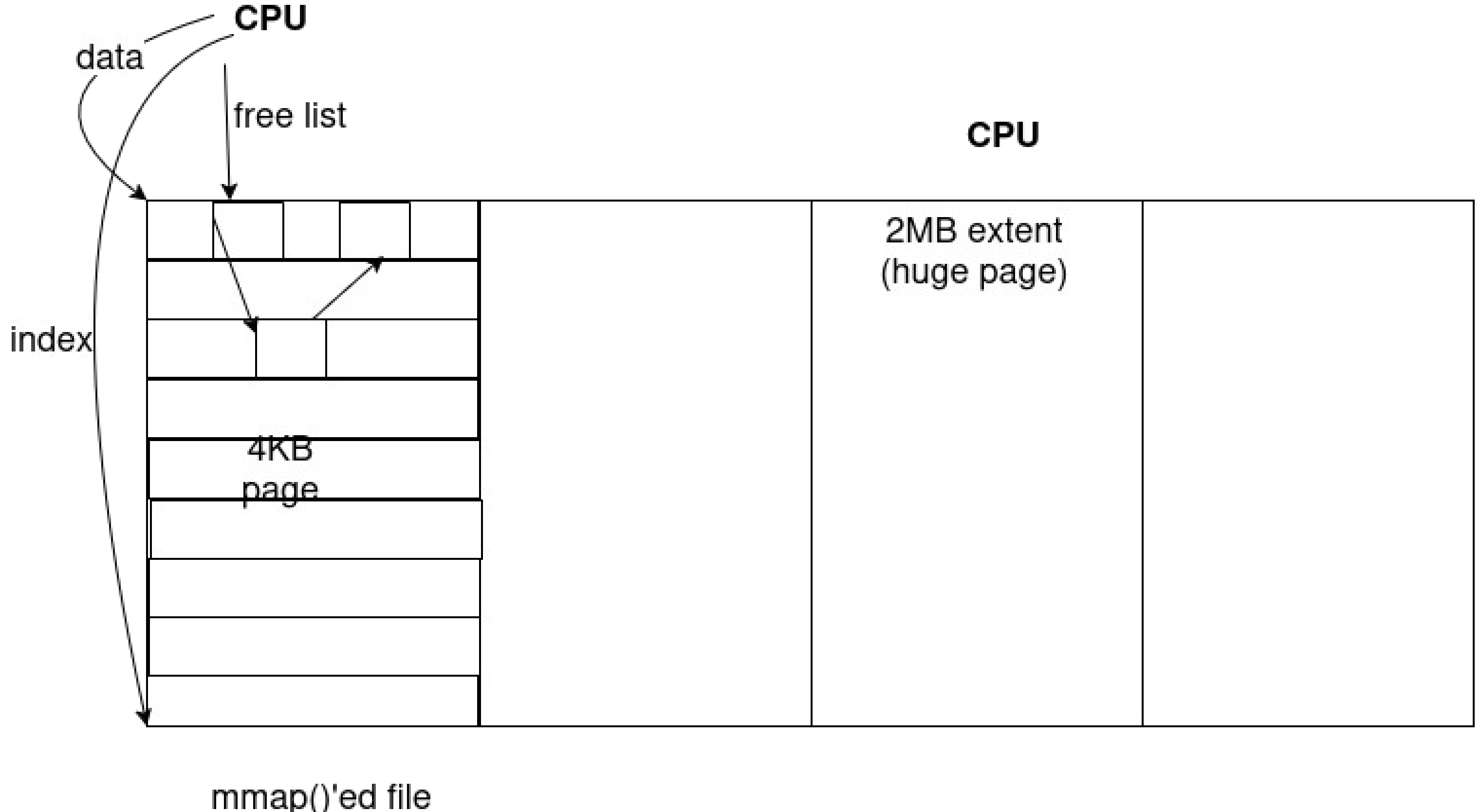
Storing data in-place vs metadata

- Data in bucket
  - 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



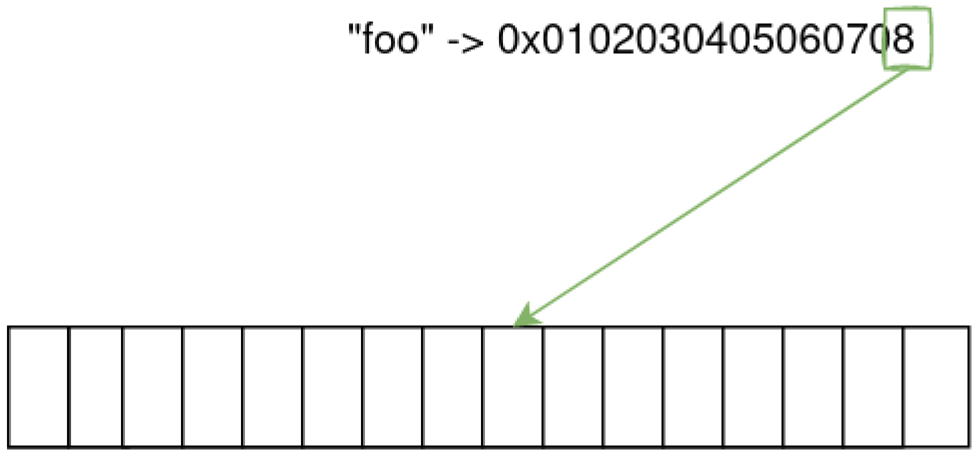
Memory allocation

- Database shard (file) is up to 128GB
- Each CPU works with local allocator within an extent
  - for small records CPUs can share 1 extent
- 2 free lists: buckets and data blocks



Index node

```
const size_t HTRIE_BITS = 4;
const size_t HTRIE_FANOUT = 1 << HTRIE_BITS;
const size_t HTRIE_DBIT = 1 << (sizeof(int) * 8 - 1);

struct HtrieNode {
    uint32_t shifts[HTRIE_FANOUT];

};
```

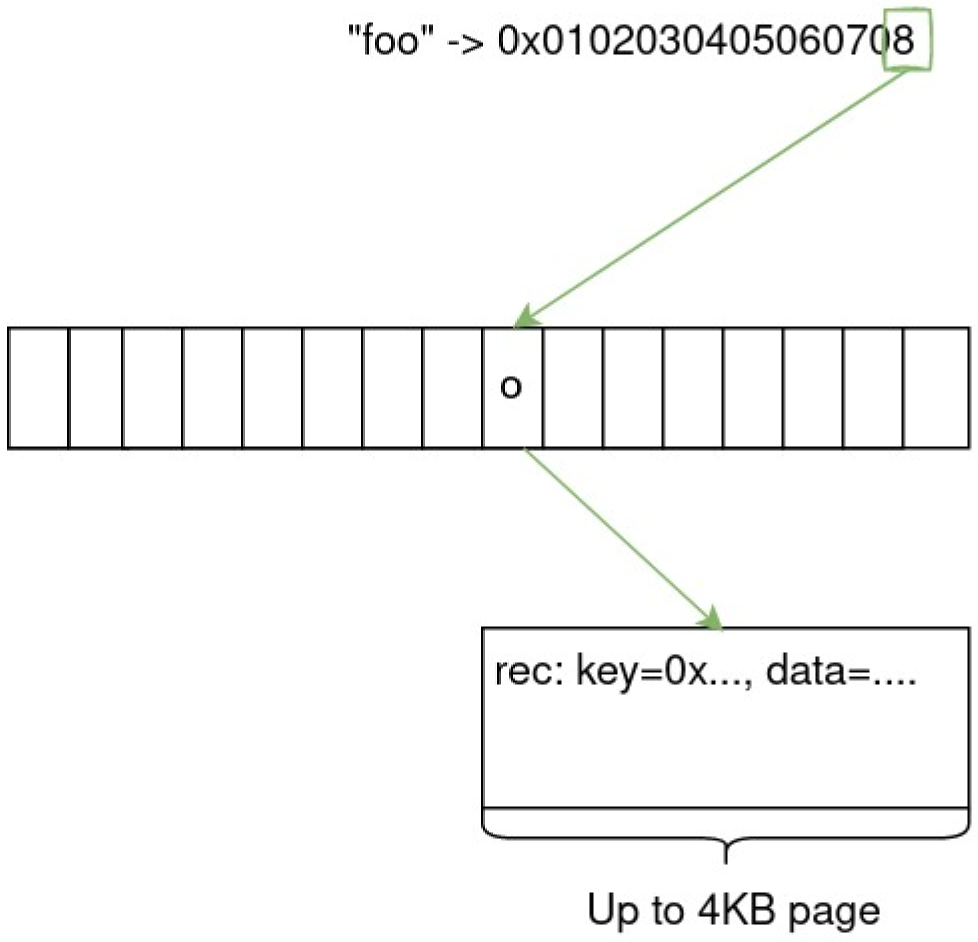


- 1 bit is reserved for “last level” (data or index block offset)
- Root node may resolve more bits (e.g. 8, 12, 16…)
- 1 index node = 1 cache line: `16 * sizeof(int) = 64 bytes`
- Maximum index: $2^{31} * 64 = 128$GB for one shard

Tempesta Technologies

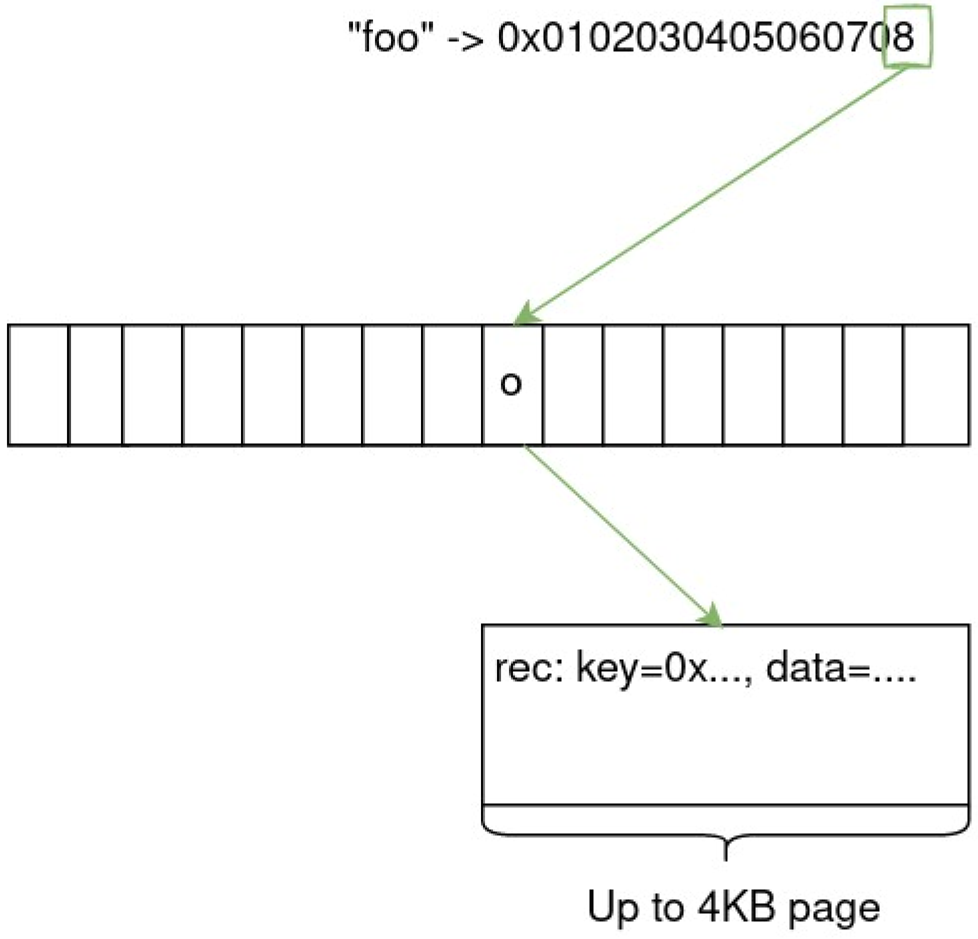
Collision bucket

```
struct TdbHtrieBucket {
    // Each slot takes 2 bits:
    //   00 - slot is empty
    //   10 - regular occuped slot
    //   01 - record removal in progress
    //   11 - record write in progress
    // (up to 32 collisions)
    uint64_t      col_map;
    . . .
};

// Acquire an empty collision slot
do {
    bm = ~(b->col_map | mask);
    if (unlikely(!bm))
        return -1;
    b_free = fls64(bm);
    if (tdb_htrie_bckt_burst_threshold(b_free))
        return -1;
} while (sync_test_and_set_bit(b_free, &b->col_map));
```



Bucket creation

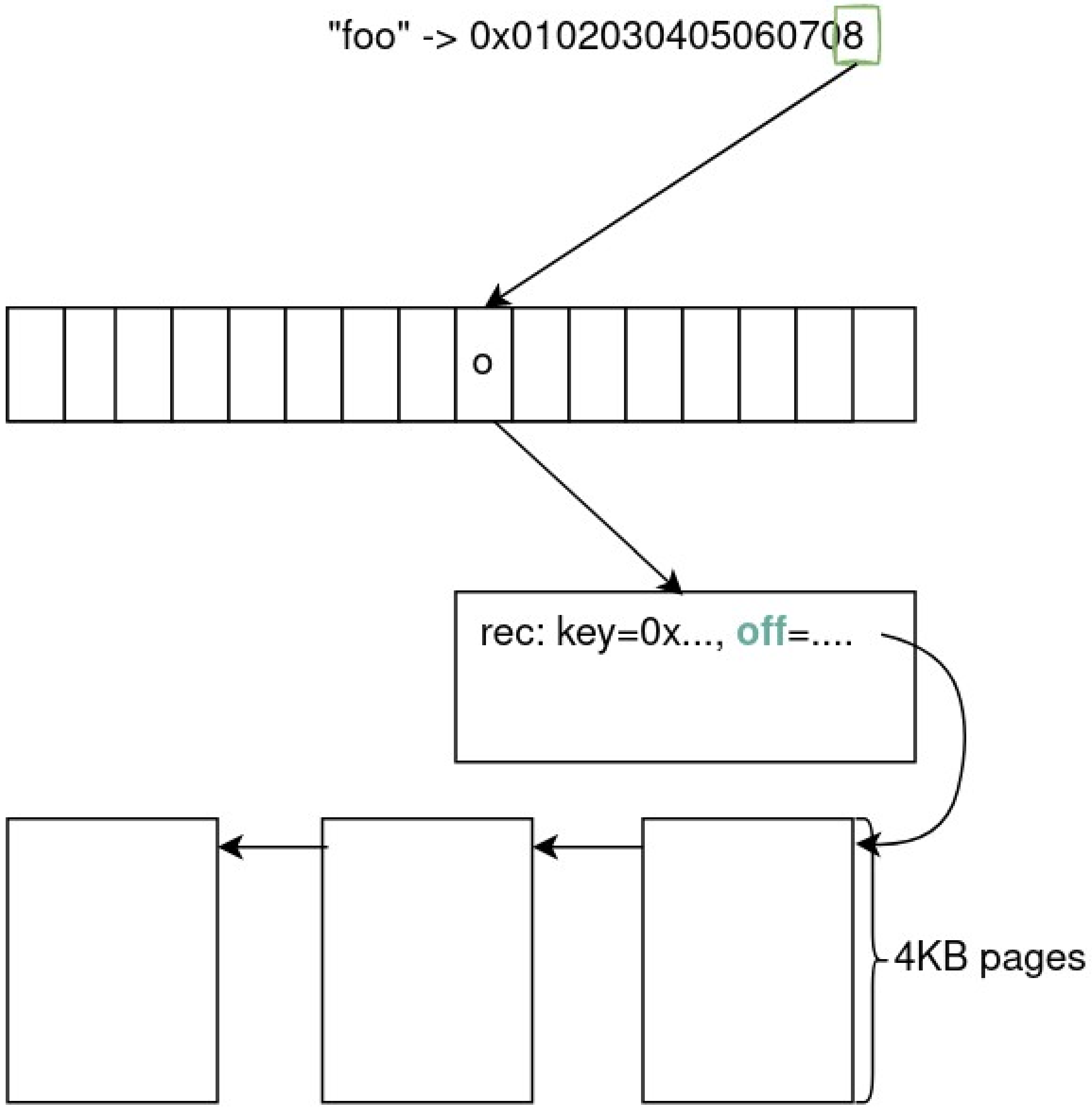
```
while (1) {
    node = htrie_descend(key);

    // 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

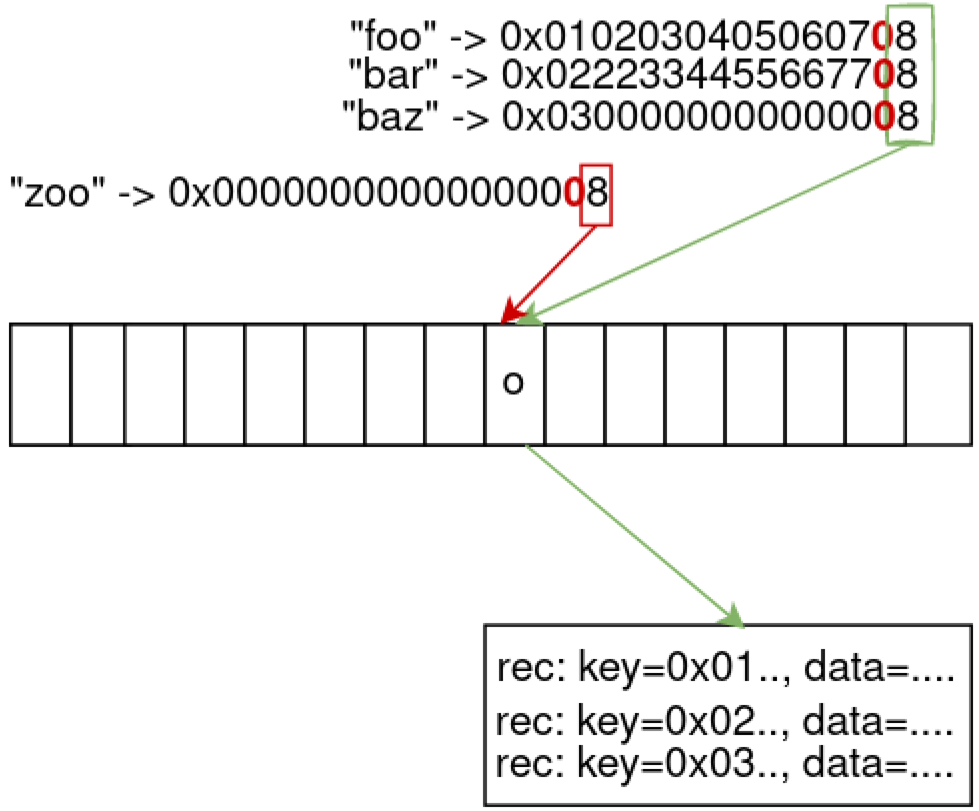
(pointer stability in bursting buckets)

```
// 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) { // key bits collision

    // 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_map);

        if (curr_map = old_map)
            break;

        map = curr_map ^ map;

        // Copy records for the new collisions

        map = curr_map;

    }
}
```



Mix hazard pointer with RCU

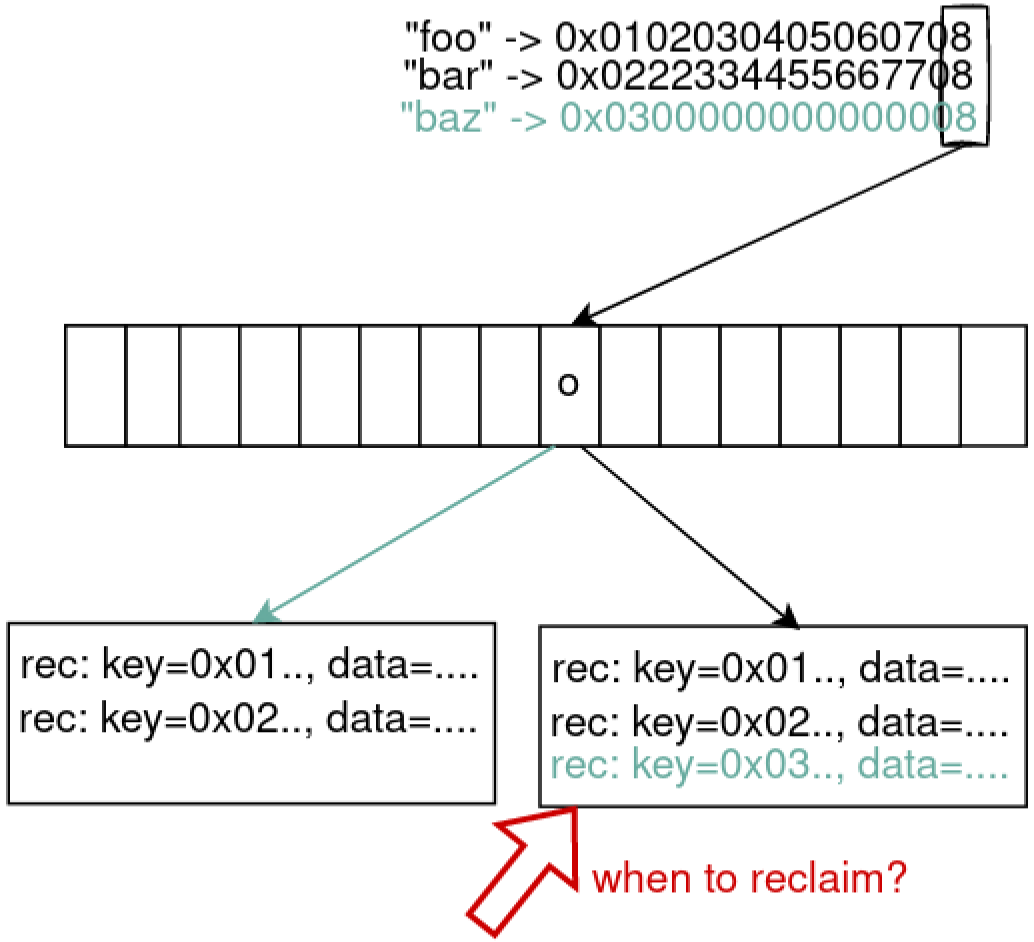
- Hazard bucket pointer: only one bucket is observed at a time
- Buckets are scanned for some time
  - keys are **large**, **compound** and **part** of stored object
- Update: **copy** a bucket, even for a record **removal**
  - Requires memory to remove a record

Tempesta Technologies

Reclaiming data

(Don’t delete empty index nodes, it’s just 64B)

```
struct TdbPerCpu {
    uint64_t  active_bckt; // hazard pointer
}

tdb_htrie_descend_get_bckt() {
    do {
        // get regular pointer to a bucket
        o = tdb_htrie_descend(dbh, key, bits, node);
        bckt = TDB_PTR(dbh, o);
        // write the per-CPU hazard pointer
        tdb_htrie_get_bucket(dbh, bckt);
        // check that the pointer hasn't been changed
    } while (node->shifts[IDX(key, *bits)] != o);
    // use the bucket pointer
    return bckt;
}

htrie_remove() {
    // copy & update the backet, CAS() the index

    // check hazard pointers on all CPUs
}
```



Tempesta Technologies

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