Efficient Data Structures via Memory Level Parallelism

Haoran Xu

Contents

1 Introduction

1.1 Background of Memory Level Parallelism

As CPU microarchitecture design continues to evolve, more and more features have been introduced to modern CPU for better performance. Some of the features, such as SIMD instructions, are supposed to be explicitly used by software, while others, such as out-of-order execution and advanced branch predictor, are designed as transparent optimizations that require no software-level cooperation.

Memory level parallelism is a feature that falls in between. It takes only a fraction of a nanosecond to execute a CPU instruction; but reading a position in the main memory takes at least 70ns even on the latest DRAM chip. While the CPU cache is supposed to hide this latency, few accesses will hit the cache if the access pattern lacks locality. Memory level parallelism was introduced in Intel CPU *Pentium 4* in 2000 to address the huge discrepancy between DRAM latency and CPU clockrate. This CPU introduces a hardware (the "hardware prefetcher") to predict the DRAM addresses that will be needed in the near future, by either speculative execution of the instruction stream, or heuristics based on patterns of recently-fetched DRAM addresses, and fetches those addresses to the CPU cache in parallel to the CPU execution. This effectively allows the CPU to read multiple DRAM positions in parallel, hence the name. Most Intel CPUs can track up to 10 outstanding DRAM accesses in parallel, with 12 for the latest CPU *Skylake* [7]. While there is no official documentation for AMD CPUs, experiment shows the latest AMD CPU *Epyc* can track at least 16 DRAM accesses. Programmers may also use a special "software prefetch" instruction to explicitly advise the CPU to fetch an address into the cache.

A large class of indexing and graph analytics applications are memory latency bounded: while the DRAM chip could support a high bandwidth, the CPU is only able to effectively use a small portion of the bandwidth, as most of the time it is stalling on memory latency. And as we will show in later sections, in most of such applications, if no proper software guidance is provided, the hardware prefetcher has very limited (actually in a lot of scenerios, zero) capability to leverage those wasted bandwidth by prefetching useful data into the cache. By effectively leveraging memory level parallelism, we could in theory allow the CPU to perform 10x or more DRAM

random accesses in the same amount of time. Therefore, memory level parallelism has huge potential to speed up those applications.

Unfortunately, despite that indexing workloads are largely bounded by memory latency, most of the prior attempts to optimize such workloads via modern programmer-usable CPU features focused on utilizing SIMD instructions, such as [8, 11, 12, 3]. Despite that memory level parallelism also provided a programmer-usable interface (the "software prefetch" instruction), there is not much prior work in utilizing memory level parallelism for better performance on indexing workloads. We summarize the prior works as well as their limitations in Section 1.2.

1.2 Prior Works and Their Limitations

To the best of our knowledge, [4] is the earliest paper that utilizes memory level parallelism for better performance. It focuses on optimizing the performance of a hash join, which is essentially processing a batch of hash table lookups on a static hash table. The paper observed that hardware prefetcher are not helpful in automatically prefetching useful data, since hash table access exhibits no predictable pattern, and the logic is too complex for the CPU to speculatively execute across multiple queries. To solve the problem, the paper divided the procedure of a hash table lookup into multiple phases, with each phase only contains one cache-missing DRAM access, which address can be computed from the previous phases. A fixed number of queries are processed together as a batch. The algorithm executes phase by phase. In phase k, it executes the phase k of all queries in the batch, and prefetches the DRAM addresses that will be used in phase $k+1$. There is one outstanding prefetch for each query at any time, and those prefetches will be executed in parallel thanks to CPU memory level parallelism capability.

As far as we are aware of, all existing methods to leverage MLP are based on the above-mentioned "software pipelining" technique. This technique can be used to support batched-read operations of a number of widely used data structures, including hash table, binary search, binary search tree, skiplist and trie tree, as shown in [9, 8]. The authors of [9] demostrated that the technique delievers up to 6.4x single-thread speedup, and 2.5x total throughput when running on all cores.

AMAC [10] is an improvement to the software pipeline technique that addresses the issue that some queries may have less or more phases than others (for example, the length of probing chain varies for each hash table lookup). Unlike [4], which operates on a fixed batch of queries, the "active set" is dynamic in AMAC. Whenever a query finished its execution, it is removed from the active set, and new queries can come in, which makes sure that a high MLP is always exposed to the CPU.

[14] applied the software pipelining technique to a wider range of database operations. Specifically, it decomposes the procedure of executing a query into stages, where each stage only performs operations on cache-resident data and prefetches. To gaurantee that there is always a high degree of memory level parallelism, the outputs of a stage are temporarily buffered. They are not sent to the next stage until a full vector of outputs (the batch) has been generated from the inputs, which gaurantees that each stage can always operate on a full batch of inputs.

In practice implementing a software pipeline correctly and efficiently often requires significant labor work, as well as extensive change to the application logic, yet the resulted code is almost always tricky, error-prone, unreadable and unmaintainable – the exact opposite to the standards of high quality industrial code. [9] addressed this issue by designing a Domain Specific Language called Cimple that generates and manages software pipeline automatically from C-like code via a coroutine technique. In Cimple , all one needs to do is to annotate DRAM accesses in the code that are supposed to be cache-missing. The authors demostrated that the automatically generated software pipelines via Cimple have comparable performance as the hand-written ones, on a variety of well-known data structures.

However, there is one fundamental limitation of software pipelining technique. Software pipelining can only support batched-read-only workloads. It is impossible to support non-batched workloads (e.g. the next query may be dependent on the answer to the previous one), since the nature of software pipeline requires a batch of queries to expose memory level parallelism. It is also extremely hard, if not impossible, to support mixed workloads or even insert-only workloads in general, due to read-write and write-write hazards similar to one would encounter in a multi-threading environment. Since a batch of reads and writes are executed in a interleaved manner in

the pipeline, a read may observe a logically earlier but not-yet-completed write in the pipeline, or a logically later write in the pipeline that has wrote part of its data. Multiple partially completed writes in the pipeline may also interfere with each other. This fundamental limitation, as well as the practical difficulty to implement and maintain a software pipeline, may explain why software pipelining is rarely used in real world.

1.3 Our Contribution

We take a completely different approach. Instead of sticking to traditional data structures and trying to leverage memory level parallelism by software pipelining multiple read queries together, we design new data structures from ground up that can leverage memory level parallelism from the inside of each query ("internal parallelism"), not across multiple queries. Since the parallelism is drawn from the internal of every query, our approach fully overcomes the limitation of software pipelining. We can support both read and write queries, and do not require batched-querying at all. Furthermore, the nature of our approach makes it transparent to higher-level application. Our data structures provide the exactly same APIs as their traditional counterparts do, so programmers can enjoy the benefits of memory level parallelism without even the need to understand what it is.

As far as we know, we are the first to design data structure that grounds its performance on memory level parallelism. A good analogy of how our data structures are grounded on memory level parallelism is how B-tree is grounded on blocked disk reads. Despite that a B-tree might not read less bytes than a conventional binary search tree, B-tree operates faster since disk reads are done in blocks. Similarily, we might not read less locations from the memory than a conventional data structure does, nonetheless we are much faster, because we are reading those locations in parallel. This "ground-up" approach turns out to be fruitful. As an example, our MlpIndex , an ordered index structure providing identical APIs as std::set, runs 3 to 4 times faster than the state-of-the-art ordered index, while the algorithm being conceptually much easier to understand.

The rest of the paper is organized as follows. In Section 2, we measure the effect of memory level parallelism on a set of real-world CPUs, In Section 3, 4, 5, we provide 3 examples of

our "MLP-grounded" data structures – a lock-free skiplist iterator, a priority queue, and an ordered index. All of our data structures have identical APIs as their traditional counterparts. In Section 3, we present how to speed up lock-free skiplist iteration via memory level parallelism. We employed our technique in MemSQL [13], a commercial main-memory database that uses lock-free skiplist as their index data structure [1], and demostrated that our technique achieves up to 1.9x single-threaded end-to-end query time speedup, and 1.5x to 1.9x total throughput when running on all cores. In Section 4, we present a priority queue based on the binary heap, but redesigned to allow internal parallelism. We demostrate that it is [todo] faster than the stl implementation [todo: comparing against stl is stupid]. Finally in Section 5, we present MlpIndex , an ordered index for 64-bit integers (and can be extended to support short strings), inspired by X-fast trie [6], but re-designed from ground up to leverage memory level parallelism. We demostrate that it can execute an insert or lookup operation in time equivalent to around 2 DRAM roundtrips, and an lower_bound operation in time equivalent to around 3 DRAM roundtrips. This is 3 to 4 times faster than HOT [3], the current state-of-the-art ordered index structures. We note that our insert and lookup performance is only slightly slower than a hash table, which cannot support range queries like lower_bound . We also note that while most of the current state-of-the-arts ordered index are based on trie tree, which makes their performance inheritally dependent on the input data distribution, the nature of our approach makes its performance independent of the input data, and we are still 2 to 3 times faster even if comparing our worst case against the best case of our benchmark rivals. In Section 6, we conclude this paper with a few directions of possible future works.

2 Understanding MLP in Real World

In this section, we measure the effect of memory level parallelism in real world. The experiment is performed on a few different hardware settings, including an Intel-CPU personal laptop (*Skylake*, DDR4), an Intel-CPU multi-socket server (*Haswell*, DDR4), and an AMD-CPU multi-socket server (*Epyc*, DDR4). However, the results are similar with only quantative difference, so we will only present the results measured on the personal laptop. The hardware setting is Intel

Core i7-7700HQ CPU at 2.80GHz and 1x16GB Corsair DDR4 2400MT DRAM.

2.1 Speculative Execution, and Stopping Speculative Execution

We first review the CPU memory level parallelism mechanism mentioned in Section 1. There are two hardware mechanisms that allow the CPU to obtain DRAM addresses to prefetch:

- Address predicted from patterns on addresses from recent accesses (hardware prefetcher).
- Address predicted through speculative execution of the instruction stream.

In order to reliably measure the effect of memory level parallelism at different "parallelism degrees", we want to be able to control the effectiveness of the two hardware mechanisms.

To control the hardware prefetcher, since it is clearly impossible to predict the next generated random number without the knowledge of the generator, we can make it ineffective by performing accesses at positions generated by a random generator, and leads to the following setup:

```
// Each element occupies one cache line (64 bytes)
struct Element {
    uint64_t value;
    uint64_t padding[7];
};
// The big array allocated at cache line boundary
// Assume SIZE is a power of 2
Element arr[SIZE];
for (int i = 0; i < SIZE; i++) arr[i].value = rand();
// The indices to access, which are random numbers in valid range
uint64_t idx[N];
for (int i = 0; i < N; i++) idx[i] = rand() % SIZE;
```

In the above setting, each array element occupies one cache line. The indices array is populated with random values, so the hardware prefetcher will not be able to see any pattern.

To control the speculative execution mechanism, we need a method that makes it work perfectly, as well as a method to totally disable its effectiveness.

It is not hard to construct a case where speculative execution works perfectly. If the instruction flow does not have any data dependency or branches, and is succinct enough to fit in the speculation window, speculative execution will just work. The simplest example is the following code, which represents the case where CPU memory level parallelism can work to its full potential:

```
for (int i = 0; i < N; i++) {
    sum += arr[idx[i]].value;
}
```

Note that despite this code looks branchy, it is actually not a problem since the compiler will automatically unroll the loop to reduce the frequency of branches, and the hardware will also be able to predict the outcome of branch correctly.

Reliably stopping speculative execution is a little more tricky. A nature attempt is to create a data dependency on the result of a previous DRAM access. One might write the following code:

```
for (int i = 0; i < N; i += 2) {
    uint64_t v = arr[idx[i]].value;
    // create a data dependency on v
    uint64_t newIndex = (idx[i+1] ^ v) % SIZE;
    sum += arr[newIndex].value;
}
```

But actually the CPU will be able to speculatively execute the next loop iterations and determine that the address arr[idx[i+2]], arr[idx[i+4]], etc, will be eventually needed, thanks to the out-of-order execution mechanism. To fully prevent speculative execution, we need to have a data dependency built upon the results of all previous accesses:

```
for (int i = 0; i < N; i++) {
    uint64_t newIndex = (idx[i] ^ sum) % SIZE;
    sum += arr[newIndex].value;
}
```

The above code represents the "worst case": the CPU is executing completely in serial, issuing down a DRAM access, wait for it, and until the access returns can the CPU know the address of the next access. We note that however, the above code is not an accurate measurement of the DRAM latency, since a significant portion of time in above code is spent in resolving the TLB miss, unless huge page is used. We will discuss in detail the effects of huge pages in 2.4.

2.2 Measuring MLP at any degree of parallelism

The two constructs in Section 2.1 allow us to in general, construct code snippets that are able to measure the memory level parallelism at any degree:

```
// Suppose we want to measure MLP at degree K
for (int i = 0; i < N; i += K) {
    uint64_t oldSum = sum;
    // The address for the K accesses below are
    // independent from each other, but dependent on
    // everything prior to this loop iteration
    for (int j = i; j < i + K; j++) {
        uint64_t newIndex = (idx[j] ^ oldSum) % SIZE;
        sum += arr[newIndex].value;
    }
}
```

Essentially in the above code, the CPU is exposed to K independent accesses which can be done in parallel. Then the results of the accesses are sent to an accumulator. That accumulator acts as a global "data dependency barrier" that fully prevents speculative execution beyond it, since every accesses later is directly or indirectly dependent on the value of this accumulator, which cannot be known until the K accesses in the current stage finishes. This allows us to measure the performance of MLP at parallelism degree K. As a side note, one can easily extend this technique to mimic any data dependency graph.

Figure 1 illustrates results for different K, ranging from 1 (serial access) to 120.

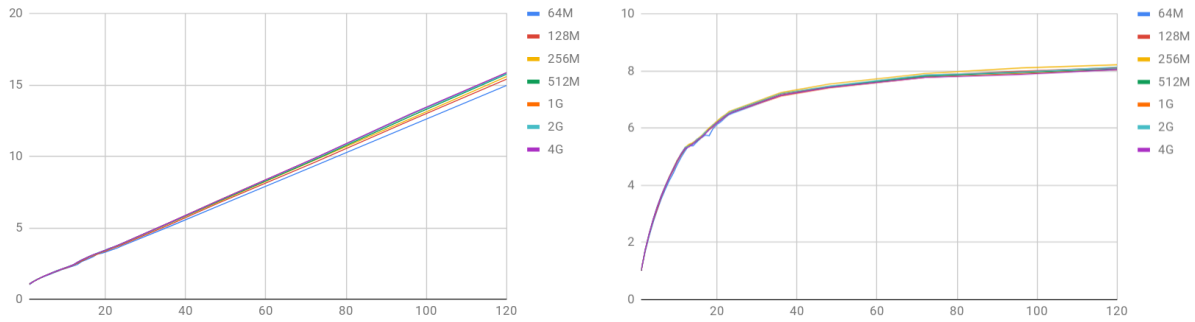

Figure 1: Left: the time it takes to complete K independent accesses in parallel, normalized against the time it takes to complete 1 access. Right: the equivalent memory level parallelism degree one can reach by issuing down K accesses at a time. Colors: different sizes of array arr.

Note that 1GB huge page is used in the experiment, see Section 2.4 for further discussion on huge pages.

As shown in Figure 1, the time it takes to perform K independent accesses in parallel is pretty much linear with respect to K. One can obtain the following empirical formula from the figure, to characterize the time it takes to perform K independent parallel accesses in terms of T, the equivalent number of serial accesses:

$$T = 1 + c \cdot (K - 1)$$

where c is a constant that depend on the specific hardware. In Figure 1, measured on the *Skylake* laptop, $c \approx 0.117$. The *Haswell* multi-socket server also has $c \approx 0.117$, while the AMD *Epyc* multi-socket server has $c \approx 0.065$, which suggests that AMD CPUs can track more outstanding DRAM accesses in parallel than Intel can.

2.3 The Effect of Adjacent-Line Prefetcher

In this section, we measure the performance of a particular hardware prefetcher – the adjacent-line prefetcher. When one accesses a cache line, the adjacent-line prefetcher will automatically prefetch its "adjacent cache line", the cache line that complements the accessed cache line in its 128-byte block, into the CPU cache.

To measure the effect of adjacent-line prefetcher, one needs to construct a code snippet that accesses the adjacent line of a previously accessed line. However, one must "hide" the address of that access from speculative execution mechanism, to make sure that we are measureing the effect of hardware prefetcher, not speculative execution. This can be done via the following trick: access a few (say 8) random cache lines first, and accumulates the results. We then access the adjacent cache line of the k-th access, where k is the residue of the accumulator modulo 8. We are always accessing an adjacent line, but which line to access is not known until all prior DRAM accesses have completed. So the best thing speculative execution can do is blindly guess the residue, which is unlikely to hit since there are 8 possibilities.

We also measured two other interesting cases: change the adjacent line access to instead access the cache line itself (which will be a L1 hit), and access the "non-complementary" adjacent line,

which is the line adjacent to the cache line, but resides in the other 128-byte block.

Suppose a DRAM access takes 1 unit of time. Experiments showed that an L1-hit takes around 0.1 unit of time, the adjacent line access takes around 0.2 unit of time, and the "non-complementary" adjacent line, somewhat surprisingly, takes 1 unit of time.

This suggests that the hardware adjacent-line prefetcher indeed allows the programmers to access the adjacent line at a relatively low cost (only twice the cost of an L1-hit). However, it is important that one aligns its 128-byte block of data along the 128-byte boundary. If one mistakenly aligned the block along a 64-byte boundary that is not a 128-byte boundary, the access to the "adjacent line" will take the full cost of 1 DRAM access time.

2.4 Tricks for Minimal Latency

We note that a few tricks are needed for best results of memory level parallelism. The most important trick is to enable huge pages. For random accesses over a huge address range, it is likely that every access is a TLB miss if the default 4KB pages are used. This turns out to have a very significant impact on the potential of memory level parallelism. Figure 2 is the result of the experiment in Section 2.2, except that the 4KB page is used instead of 1GB huge page.

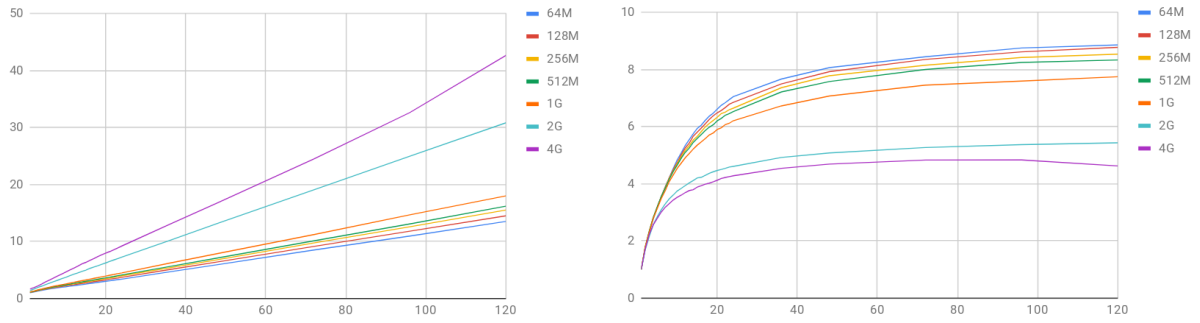

Figure 2: Experiment of Figure 1, but run with default 4KB pages.

As one can see, the performance quickly degenerates as the array size becomes larger. On 4GB array, we could only achieve MLP slightly over 4, compared with 8 in the 1GB page case. 2MB huge page turns out to be sufficient as well: it is slower than 1GB pages, but only slightly.

Multi-socket servers may need additional tricks for optimal performance. First, the thread should be pinned to a fixed CPU, as otherwise it may be migrated to another socket by the

kernel, then all DRAM accesses would be on the other NUMA node, which will be significantly slower. Second, due to cache coherency issues, every access to DRAM needs to consult the caches of the other CPU socket, to make sure it is not concurrently modified by the other CPU socket. However, if the other CPU socket is idle, it will run at a much lower frequency, which will make answering those cache coherency requests slower. So one should run an infinite loop on the other CPU socket to keep it busy and run at full frequency.

Filling more DRAM slots on the motherboard is also helpful for best performance. All plots presented above are run with only 1 DRAM chip, leaving the 3 other slots on the motherboard empty. If one plugs in 2 DRAM chips, one can actually get ∼10% better results in high MLP cases. This might be caused by contention on the DRAM chip when MLP is high.

3 Lock-free Skiplist Iterator

Lock-free skiplist is a popular choice of index data structure for main memory databases. One of its most important use cases is to support range query: given a range on the index key, one wants to iterate through every item which index falls in the range in increasing order. There are often at least hundreds of items falling in the specified range. In that case, iterating through every item in the range will take the majority of the time, compared with the $O(\log n)$ time to find the boundary nodes corresponding to the ranges. The nature of skiplist decided that every iteration will be a cache miss.

In this section, we describe how to speed up this iteration process. Our iterator provides the identical API as the plain iterator – specifically, the "++()" method that advances the iterator to the next node, and plain pointer dereference to access the current node. This makes us a drop-in replacement of the plain iterator for higher-level applications. In Section 3.1, we elaborate the idea on a single-threaded skiplist, and demostrate a 4x speed-up. In Section 3.2, we extend our iterator to support lock-free versioned skiplist, and integrated our iterator into MemSQL [13], a commercial main memory database, and evaluated the performance impact on real SQL queries. We demostrated that our iterator results in 1.9x end-to-end query time speedup when there is only one query thread. With multiple query threads, we still achieve 1.5x to 1.9x speedup.

3.1 Single-threaded Skiplist Case

We first briefly review how skiplist works. A skiplist is essentially a multi-layer linked list. The bottom-most (0th) layer is a linked list of all items, in sorted order. For each $k \geq 0$, layer $k+1$ is a linked list of a subset of items in layer k, sampled with a probability of $1/2$, in sorted order. This results in a so-called skiplist tower, as shown in Figure 3.

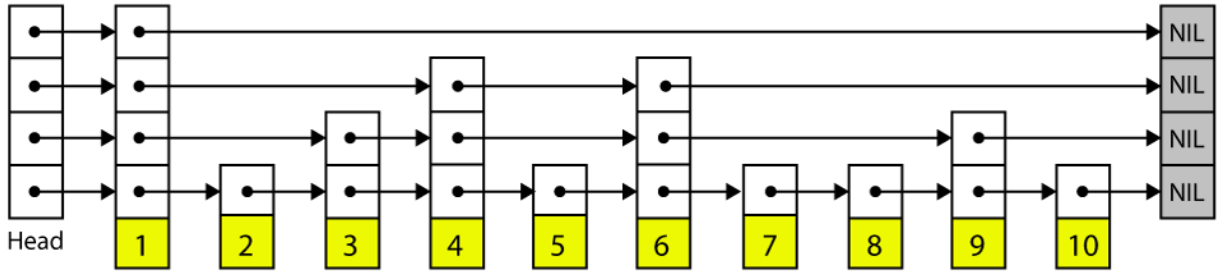

Figure 3: An illustration of a skiplist tower, from [1]

Since each layer is a sampling of its next layer with probability $1/2$, there are $O(\log n)$ layers in expectation for a skiplist containing n items, and there are expected $O(1)$ items in the next layer between two adjacent nodes of any layer. This allows one to answer a lower_bound query in expected $O(\log n)$ time by traversing from the start of top layer, moving along the linked list as long as the next node is smaller than the given bound, and moving down to the next layer when the condition is not satisfied. After the two boundary nodes corresponding to the specifed range are found, iterating through every node in this range is no different from iterating a linked-list: one simply traverse along the bottom-most layer starting from the left boundary node, until the right boundary node is reached. While the data structure is very straight-forward, for clearity, we still provide the prototype of the skiplist as well as a plain iterator:

```
struct Node {
    // the payload, which we don't care
    PayloadType payload;
    // height of tower of this node
    int height;
    // an array of length height, next[k] is next node in layer k
    Node* next[];
};
```

```
struct Iterator {
    Node* current;
    // Access current position
    Node& operator*() { return *current; }
    // Advance to next position
    Iterator& operator++() {
        // Move to next node along bottom-most linked-list
        current = current->next[0];
        return *this;
    }
};
```

Interestingly, one can easily show that it is impossible to leverage MLP from linked-list iteration: the address of current.next.next will not be known, until current.next is fetched from memory. This is exactly the "worst case" in Section 2.1 where each access (directly or indirectly) depends on every access before it, so the plain iterator has MLP=1.

How could we get more MLP? The trick is, the skiplist tower gives us a way to know about nodes further ahead: current.next[k] will give us a node approximately 2^k steps away from current node. Of course, this only works if the tower height of current is at least k.

So a natural idea to issue down more than one independent accesses at once is to use the level 1 tower. Whenever we encounter a node c with height larger than 0 (which happens with 50% probability), we will prefetch c.next[1]. If c.next[1] is not equal to c.next[0] (which happens with 50% probability), then we prefetched something useful. This allows us to have MLP=2 with 25% probability, and MLP=1 otherwise, resulting in an overall MLP of 1.25, as shown below.

```
// Naive prefetch strategy using level 1 tower
Iterator& operator++() {
    // Prefetch level 1 tower if useful
    if (height > 0 && current->next[0] != current->next[1]) {
        __builtin_prefetch(current->next[1]);
    }
    // Move to next node along bottom-most linked-list
    current = current->next[0];
    return *this;
}
```

We can extend the idea to do better by using the higher tower levels as well. We will have a queue of "prefetch tasks", each described by a triplet $(start, end, level)$. The triplet means that

start has been prefetched to cache (so accessing *start*.next[] will not be cache-missing), we want to prefetch everything between *start* and *end*, and the highest layer linked-list that connects *start* and *end* is *level*. To process such a task while maintaining the invariance, we will:

- Prefetch *start*.next[i] for each $0 \leq i \leq level$.
- Add a task to traverse from *start*.next[i] to *start*.next[$i+1$] via level i, for $0 \leq i < level$.
- Add a task to traverse from *start*.next[*level*] to *end* via *level*.

This gives us the following pseudo-code:

```
struct QueueItem {
    Node* start; Node* end; int level;
};
// Process a prefetch task, and enqueue generated tasks into q
void ProcessQueueItem(QueueItem task, queue<QueueItem>& q) {
    // Step 0: check termination case
    if (task.start == task.end) {
        return;
    }
    Node* n = task.start;
    // Step 1: prefetch next node in each layer
    for (int i = 0; i <= task.level; i++) {
        __builtin_prefetch(n->next[i]);
    }
    // Step 2 & 3: put generated tasks into q
    for (int i = 0; i < task.level; i++) {
        q.Push(QueueItem(n->next[i], n->next[i+1], i));
    }
    q.Push(QueueItem(n->next[level], task.end, task.level));
}
// Process a whole queue of tasks, returns new queue of tasks
// Note that all accesses in this function are cache-resident,
// and all prefetches are independent and executed in parallel
void ProcessQueue(queue<QueueItem>& q) {
    queue<QueueItem> newQueue;
    while (!q.Empty()) {
        ProcessQueueItem(q.Pop(), newQueue);
    }
    q = newQueue;
}
```

Analyzing the code above, it is not hard to see that for a task with level k, it will be fully "digested" in k steps, while producing on expectation C_k^i non-empty ($start \neq end$) items in the

i^{th} step. So if we push a new level K task into the queue in every phase, the queue will stablize at expected $1 + \sum_{i=1}^{K} C_K^i = 2^K$ items, with expected 2^K useful prefetches issued in parallel in each phase. This allows us to reach an arbitrary degree of memory level parallelism in theory by picking a large enough K. In practice, $K = 3$ or 4 yields the best results, which is a result of balance between larger constants hidden in the algorithm and higher MLP, and the fact that Intel CPUs can in theory have a maximal MLP of 10. The pseudocode can be found below.

```
// Max tower level to be used
const int K = 3;
struct Iterator {
    Node* current;
    // The prefetch queue, size 2^K on expectation
    queue<QueueItem> q;
    // The furthest node reached and prefetched
    // It must be a node of height at least K
    Node* furthest;
    // Issue one phase of prefetches
    // On expectation 2^K parallel prefetches will be issued
    void Prefetch() {
        // Process each item in the queue
        ProcessQueue(q);
        // Finally add a new level K task into queue
        q.Enqueue(QueueItem(furthest, furthest->next[K], K));
        // Advance and prefetch furthest node
        furthest = furthest->next[K];
        __builtin_prefetch(furthest);
    }
    // Advance iterator
    Iterator& operator++() {
        // Start a new phase of prefetching when needed
        while (current == q.Head().start) {
            Prefetch();
        }
        // Move to next node along bottom-most linked-list
        current = current->next[0];
        return *this;
    }
    // Setup prefetch queue on construction
    Iterator(Node* current) : current(current) {
        // Find the first node n with height >= K by brute force
        Node* n = ....
```

```
            // Setup prefetch queue and the "furthest" node
            q.Push(QueueItem(n, n->next[K], K));
            furthest = n->next[K];
            __builtin_prefetch(furthest);
        }
    }
```

We benchmarked this improved iterator against the plain iterator. Experiments showed that for long iterations of a few hundred items, we are 4x faster than the plain iterator. Since our iterator has a ramp-up cost (we need to find a node with at least tower level 3, and for the first few phases our queue is not having the full expected size thus less MLP, and for the last few phases we may prefetch items that are outside range), for short iterations our advantage is smaller. However, we are advantagous as long as the range contains at least around 20 items.

3.2 Integration into MemSQL, a Real World Database

In this section, we demostrate the performance impact of our iterator in MemSQL [13], a commercial main memory database.

MemSQL uses a versioned lock-free skiplist as their in-memory index, with each node represent a row of data. Versioning is necessary to support snapshot isolation. Specifically, each lock-free skiplist node stores a lock-free linked-list of all versions of the data in that row. To find the row that correspond to a specific snapshot version, one needs to iterate the linked-list to find the largest version that is smaller than the given snapshot version. So we need to prefetch not only the next nodes, but also the head of the version linked-list of the current node. There is no optimistic locking protocol: updates are supposed to be sparse. As such, it is plausible to assume that most of the time the version linked-list will only contain one element, so we only prefetch the head of the linked list and not the deeper nodes.

Lock-freedom is generally not a problem for us: lockfree skiplist implementation gaurantees that at any point each layer is always a valid linked list even in the presence of deletes, and the memory are reclaimed using epochs, so the references we hold will also remain valid all the time. There is only one issue with delete: the “while (current == q.Head().start)” line in advance iterator may never be true, if the q.Head().start were physically in the skiplist when it was pushed

into queue, but deleted and physically removed from the skiplist by concurrent threads later. The solution is to instead simply execute the prefetch phase every 2^K advances. While this is less "intelligent" than the single-threaded version because we are no longer "adapting" to the actual tower, nonetheless it works.

We replaced the original iterator (which is just the plain iterator) in MemSQL with our iterator. To remove unnecessary overhead from distributed system, the benchmark is done under the internal SingleBox mode where all sharding and replication components are disabled. We benchmark the performance impact of our iterator with the following table schema:

```
CREATE TABLE t (a INT, b CHAR(16), KEY(a));
```

We populate 20 million rows of random data into the table, and time the following SQL query:

```
SELECT COUNT(*) FROM t WHERE a > @
```

This allows us to compute the number of rows iterated per second. The original MemSQL could scan 5.0M rows per second. After switching to our improved iterator, we are capable of scanning 9.6M rows per second, a 1.92x speedup. We also tested the case of 4/8 concurrent query threads working on 2/4 available CPUs, the results are in Table 1. As one can see, we achieve 1.54x to 1.93x speedup in multi-threaded cases.

Query Threads MemSQL CPU	4	8
4	19.8M vs 30.5M (+54%)	16.8M vs 27.7M (+65%)
2	12.0M vs 23.2M (+93%)	12.8M vs 20.9M (+63%)

Table 1: Throughput for multi-threaded cases, in form of Original vs Ours (Speedup Percentage).

The reason that the speedup is smaller compared with the single-threaded case may come from a few factors. First, the versioned lock-free skiplist contains significantly more logic than the single-threaded skiplist. We cannot speed up those logics, and furthermore, some of the logics contain DRAM access (for example, reading the version linked-list), which consumes some of the fixed hardware-available memory parallelism degree. Second, versioned lock-free skiplist contains a lot of memory barriers. Those barriers could require flushing the CPU load/store

queues, which limits the memory level parallelism. Finally, MemSQL did not use huge pages. As shown in Section 2.4, this has a negative impact on the performance of memory level parallelism.

4 Priority Queue

5 Ordered Index

In this section, we present MlpIndex , an ordered index structure that supports identical APIs as std::set. Specifically, it supports insert (insert an element into the set), lookup (query if an element exists in the set) and lower_bound (find the minimum element in the set no smaller than the given element). MlpIndex is not a comparison-based data structure, but an integer data structure that operates on a key universe $[0, M)$. While our implementation currently only supports 64-bit keys ($M = 2^{64}$), it could be extended to support any short string keys as well.

MlpIndex takes a "MLP-degree" parameter D, which represents the capability of the hardware to track parallel DRAM requests, analogous to that B-tree takes a "block size" parameter that characterizes the granularity of each disk read by the hardware. Under the assumption that D independent parallel DRAM requests can be completed in one roundtrip, MlpIndex takes $O(\log_D \log M)$ DRAM roundtrips (which is the dominating factor of time in practice) and $O(D \cdot \log_D \log M)$ CPU work to execute an insert , lookup or lower_bound , and has a memory consumption of $O(n \cdot \log_D \log M)$.

To the best of our knowledge, MlpIndex is practically the fastest ordered index. It is 3 to 4 times faster than Height Optimized Trie (HOT) [3], the state-of-the-art index published in 2018. On a data set of 80 million elements, MlpIndex only takes around 2 DRAM roundtrip time to perform an insert or lookup , and around 3 DRAM roundtrip time to perform an lower_bound . We note that even a cheater who can somehow know the address where the answer to a query is stored for free will still need 1 DRAM roundtrip to read that answer.

MlpIndex is also optimal from a complexity theory view: as proven in [2], under word-RAM model and plausible assumptions on the size of the data structure, $O(\log \log M)$ per operation is the lower bound to support the operations of an ordered index.

The core idea of MlpIndex is from x-fast trie [6], a data structure mostly considered in theory. X-fast trie supports $O(\log M)$ insert , $O(\log \log M)$ lookup and lower_bound , while using $O(n \log M)$ memory. While its $O(\log \log M)$ lookup performance is theoretically optimal as proven in [2], its huge memory consumption as well as huge constants hidden in the algorithm made it not considered useful in practice. Indeed, the algorithm proposed in the original paper requires 64 hash table nodes to store a 64-bit key, and each hash table node needs to store the information of 4 pointers and 5 keys. That's already 4.5KB in space (550x larger than the original key size), and we haven't even considered the load factor of the hash table yet. While the authors proposed y-fast trie, an improvement over x-fast trie that improved the insert performance to $O(\log \log M)$ and space consumption to $O(n)$, the improvement is mainly theoretical and introduces even larger constants hidden in the big-O notation. We are not aware of any attempts to implement x-fast trie or y-fast trie as a practically usable data structure.

We did not use the ideas in y-fast trie. Our insertion performance already matches that of y-fast trie, and while our memory consumption is worse than y-fast trie in big-O notation, we would like to note that the constants hidden in the big-O is much smaller. Specifically, in practice, we use at most 96 bytes (and can be further reduced to 36 bytes) to store a 8-byte key. This is competitive with state-of-the-art index structures like ART [11] (at most 60 byte per 8 byte key) and Masstree [12] (∼58 bytes per 8 byte key as reported in [3]). As such, we deem the use of the ideas in y-fast trie to further reduce memory as possible but unnecessary.

In Section 5.1, we briefly introduce the original idea of x-fast trie from [6]. In Section 5.2 and 5.3, we elaborate the algorithm-level and engineering-level re-designs on the original idea that leads to MlpIndex , and in Section 5.4 we benchmark the performance against a few state-of-the-art ordered index implementations.

5.1 Original x-fast Trie Idea

X-fast trie assumes that all keys have a length of $\log M$ bits. In its essense, a x-fast trie is a plain binary trie that carries a few extra information on the node:

- Prefix: the prefix that this node represents in the trie.

- Min/Max: records the leaf corresponding to minimum/maximum key in this subtree.
- Prev/Succ: if the node is a leaf, records the prev/succ leaf (the "leaf-link").

Those information are dynamically updated using the information stored in its 0/1 children, for example, if one of the children of node n is modified, one can update the Min field of n using the logic described in the following straight-forward pseudocode:

```
if (n->IsLeaf) {
    n->min = n;
} else if (n->child[0] != nullptr) {
    n->min = n->child[0]->min;
} else {
   n->min = n->child[1]->min;
}
```

Furthermore, all the trie nodes are put into a hash table, indexed by the "Prefix" field of the node. This allows one to directly locate a node that corresponds to a given prefix in $O(1)$, without the need to walk through the path to that node from the root.

The core operation of x-fast trie is QueryLCP (LCP stands for longest common prefix): given a key x, find the length of the longest prefix x, such that the prefix corresponds to a node in the trie. The most naive way to support QueryLCP is to just start from the root and move down according to the bits of x. When a child does not exist, the height of the current node is the desired result. This takes $O(\log M)$ per query.

The hash table allowed us to do better. One can easily see that "whether a prefix exists in the trie tree" is monotonic with respect to the length of the prefix: suppose that the answer is k, then every prefix of length $0 \le i \le k$ exists, but any prefix of length $i > k$ does not. So we can binary search on that length and query the hash table to know if a prefix exists in $O(1)$. This reduces the time complexity to $O(\log \log M)$.

We can support lower_bound in $O(\log \log M)$ based on QueryLCP . Suppose the LCP has length k and the node of LCP is n. Since n is the LCP, n must not have child $x[k]$, as otherwise there will be a longer common prefix. However, n must have child $1 - x[k]$, since n is the prefix of some key in the trie. If $x[k] = 0$, then x is smaller than everything in this subtree, so the desired lower bound of x is the minimum node in this subtree (n.min). If $x[k] = 1$, then x is larger than

everything in this subtree, so the lower bound of x is the smallest node larger than the maximum node (n.max.succ). This results in the following pseudocode:

```
Node* Lower_Bound(Key x) {
    int len; Node* n;
    (len, n) = QueryLCP(x);
    if (len == logM) {
        // the key exists in the trie
        return n;
    }
    if (x[len] == 0) {
        return n->min;
    } else {
        return n->max->succ;
    }
}
```

Performing an insert operation is very similar. After finding the LCP node n, one just needs to build a path of new nodes to the leaf from n, and update the extra-information fields of all the nodes from the root to the newly inserted leaf, which takes $O(\log M)$:

```
// Insert a key into the trie, returns if the insertion took place
bool Insert(Key x) {
    int len; Node* n;
    (len, n) = QueryLCP(x);
    if (len == logM) {
        // the key exists in the trie
        return false;
    }
    // Find the lower_bound of x, for updating leaf-link later
    Node* succ = Lower_Bound(x);
    // Construct nodes all the way to the leaf
    for (int i = len; i < logM; i++) {
        n->child[x[i]] = new Node(....);
        n = n->child[x[i]];
    }
    // Update leaf-links
    n->succ = succ; n->prev = succ->prev;
    succ->prev->succ = n; succ->prev = n;
    // Update fields such as Min/Max all the way to the root
    while (n != root) {
        n = n->parent;
```

```
            UpdateFields(n);
        }
        return true;
    }
```

5.2 The Core Idea of MLPIndex

In this section, we present the two core ideas that build up the theoretical aspect of MlpIndex :

• Use D-ary search to replace binary search in QueryLCP for high memory level parallelism. This reduces the DRAM roundtrips required per lookup and lower_bound query from $O(\log \log M)$ to $O(\log_D \log M)$.

• Support path compression for lower insertion cost and memory consumption. This reduces insert cost from $O(\log M)$ to $O(\log_D \log M)$ DRAM roundtrips and $O(D \cdot \log_D \log M)$ CPU work, and reduces memory consumption of the data structure from $O(n \cdot \log M)$ to $O(n \cdot \log_D \log M)$.

Using D-ary search

In x-fast trie, we use a binary search to find the LCP between a given key and the set of keys in the data structure. While theoretically optimal, it did not make use of memory level parallelism, since we could not know if the answer is falling in the left half or the right half of the interval, until the current query is complete. A natural modification is to use D-ary search instead. We split the current interval $[l, r]$ into D parts, and query length $l + (r - l)/D$, $l + (r - l)/D * 2$, $\cdots$, $l + (r - l)/D * (D - 1)$. The DRAM accesses from those hash table queries are independent and can be executed in parallel. We can then narrow down to an interval that is $1/D$ size of the original. This reduces the DRAM roundtrips required per lookup and lower_bound query from $O(\log \log M)$ to $O(\log_D \log M)$.

Supporting path compression

A plain trie storing n keys of length L can have $O(nL)$ nodes. Path compression is a technique that reduces the number of nodes by contracting nodes with only one child (except for the root node, which may only have one child). Any node in a path-compressed trie is either the root node, or a leaf node, or an internal node with at least 2 children. It is easy to see that for a

path-compressed trie with n leaf nodes, there are at most n internal nodes, so a path-compressed trie of n keys can have $2n$ nodes at most.

However, path compression does not work directly in x-fast trie, because it breaks the property that binary search relies on. Let's say, the original trie has a path $root \to a \to b$, and a is removed due to path compression. Then one will see that prefix length $0, 2$ exists in the hash table, but $1, 3$ does not. Binary search no longer works.

So we have to do an "incomplete" path compression. When we perform a path compression, for each D-ary search depth d, we must create a intermediate node at the largest prefix lengths where search d will check, so that the search will see that node. Since the D-ary search takes $\log_D \log M$ steps to complete, now we need to create at most $O(\log_D \log M)$ intermediate nodes for each new node inserted, so the trie may contain at most $O(n \cdot \log_D \log M)$ nodes.

While the above point-of-view explains the memory consumption straightforwardly, it may be easier to understand the algorithm conceptually by looking at an equivalent algorithm built in a recursive way. We build a path-compressed trie tree of depth D, with each node has a fanout of $2^{\log M/D}$, and the same data structure is used recursively to maintain the list of children of each node. The special case is the root node with only one child – this correspond to the "intermediate" nodes due to incomplete path compression, which only has one child. We should not create further intermediate nodes for those intermediate nodes, so we need to treat them specially – since it can only happen on the root node, we can just store the key directly in that case. To find the LCP, we simply query every depth of the trie. Since we qeury every depth, path-compression is not a issue for us. After we find the LCP node n, we recurse to the children of n to find the LCP inside the children of n. In each step we query D independent hash table positions, and the recurse ends in $O(\log_D \log M)$ steps, resulting in a complexity of $O(\log_D \log M)$ DRAM roundtrips and $O(D \cdot \log_D \log M)$ CPU work, as shown in the pseudocode below (in which it assumes for simplicity that the length of key is a power of D).

```
// Recursive QueryLCP on path-compressed trie
int QueryLCP(Tree* t, Key x) {
    // termination case (reached last level):
```

```
    if (x.len == 1) {
        return 0;
    }
    // termination case (node with only 1 child):
    // This can only happen on the root of a subtree,
    // every other node must has >=2 child due to path-compression
    if (t.IS_SINGLE_ELEMENT) {
        return lcp(x, t.ELEMENT);
    }
    int depth = 0;
    Tree* subtree = t.root;
    // Query every depth of the trie to find LCP
    // Those queries are independent
    int step = x.len / D;
    for (int i = 1; i < D; i++) {
        int prefixLen = i * step;
        if (t.hashTable.Find(x[0:prefixLen])) {
            depth = i;
            subtree = t.hashTable.Find(x[0:prefixLen]);
        }
    }
    // Recurse to find the LCP in the children of that node
    int start = step * depth;
    return start + QueryLCP(subtree, x[start:start+step]);
}
```

Inserting into this data structure is also straightforward conceptually. One first insert the "splitting point" node if it were a contracted node before, then insert the key as a new child of the splitting point node, which is recursively inserting into the sub-data-structure that manages the children of the splitting point, as shown in Figure 4.

We additionally note that, if we apply this idea with $D = 2$ as in original x-fast trie, the data structure can be viewed as a special form of van Emde Boas tree [17] with a per-node hash table. The "path-compression" serves exact purpose as the trick in van Emde Boas tree that stores the "max node" separately, so only buckets containing at least 2 elements will build a sub-data-structure (or in the language of path compression, non-root nodes with only 1 child are contracted, and root node with only 1 child are specially treated), So from a theoretical aspect, MlpIndex can also be viewed as a hybridization of x-fast trie and van Emde Boas tree.

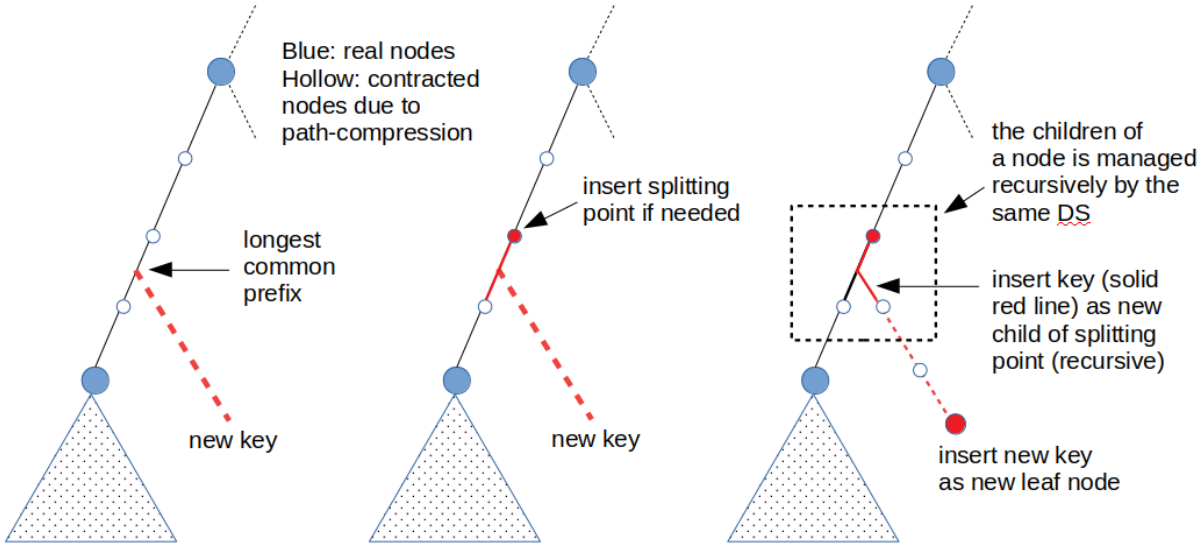

Figure 4: Illustration of inserting a key into MlpIndex

5.3 The Fully-Fledged MLPIndex

In this section, we present the ideas and tricks that are important to make MlpIndex extremely fast in practice. While the ideas are generally applicable, we focus our examples on 64-bit key case, which is the case that we benchmark in Section 5.4.

Cuckoo Hashing

The bottleneck of MlpIndex is the parallel hash table lookups, specifically, the longest probe chain of the D parallel lookups determined the number of DRAM roundtrips needed to complete this query. So we want a hash table that not only optimizes the average, but also the variance of a lookup. Cuckoo hashing [15] turns out to be the ideal choice that achieves both goals. In cuckoo hashing, any existent key x must exist in either position $f_1(x)$ or position $f_2(x)$, where f_1 and f_2 are the two independent hash functions used for the hash table. This allows worst-case 2 lookups per query, and even better, all DRAM requests issued are independent.

The original cuckoo hashing scheme allows a maximum load factor of 50%. However one can improve to a load factor of 90%+ with bucketized cuckoo [16]. Due to time limitation, we are unable to implement the bucketized cuckoo fully for MlpIndex . However, we implemented the

out-of-box experiment that demostrated the memory consumption we can achieve with a modified bucketized cuckoo, which we will discuss in paragraph 5.3.

Lowest Level Bitmap

In MlpIndex , we recurse to a subset of bits after a parallel hash table lookup. When the number of bits are small enough (say, ≤ 8), we may use a bitmap to directly store which children are existent. Note that a bitmap is sufficient and we do not need to store the pointers to the children at all – knowing if a child exists or not is enough, as we can always index to it via the hash table.

The bitmaps are small enough to be embedded directly into the hash table payload and fetched into the cache as the hash table entry is fetched, without using a pointer indirection. This allows us to incur no cache miss when querying in the last level of the data structure. This optimization turns out to have a large impact on memory consumption (which is obvious) and performance. On 64-bit keys with $D = 8$, we will only have 2 layers of the data structure. This trick allows us to work completely on cache-resident data when querying the second level, so we effectively reduced the time by a factor of two for 64-bit case.

After applying this optimization, the MlpIndex structure for 64-bit keys is effectively a path-compressed trie tree of fanout 256, with a 256-bit-long bitmap in each node to denote whether a child exists. To find the lower_bound of a given key x, we first find the LCP node n and LCP length k (in bytes) in the trie by lookup the 8 prefixes of x in the hash tables. Then, we find the lower bound of $x[k]$ amongst the children of n in the bitmap. If the lower bound exists, the answer is just the minimum element stored in the corresponding subtree. If not, we may move back to the parent of n and repeat. This allows us to remove the max-key field as well as the leaf-links and save more memory. And actually we did not sacrifice performance, since moving along the leaf-link results in a costy cache miss, but the above algorithm operates on cache-resident data all the time, and most of the time in practice we don't have to move along the parent path for more than a few steps.

Hash Table with Variant-size Payloads

A 8-bit-children bitmap takes 32 bytes to store, which is still kind of a waste if there are only a few children. We employ the "adaptive sized node" idea from [11] to further reduce memory. We demostrate how this works on 64-bit keys as an example.

Recall that the necessary information of a node includes the prefix (index) that this node represents in the trie, the path-compressed string between this node and its parent, and the minimum value stored in this node's subtree, as shown in Figure 5.

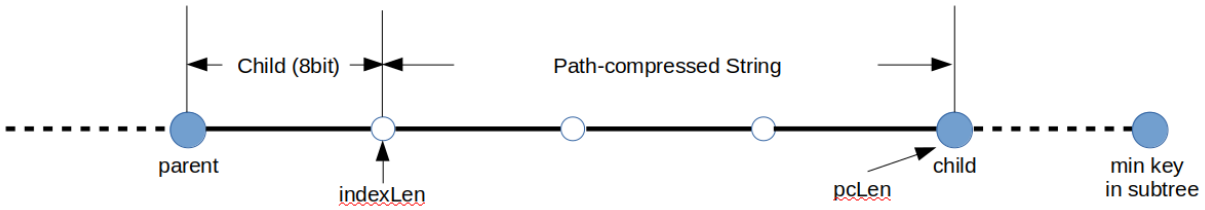

Figure 5: The trie node structure for MlpIndex on 64-bit keys.

One note that the "index" and the "prefix including path-compressed string" is always a prefix of the minKey, so we only need to store the minKey as well as the lengths. For 64-bit keys, we need around 12 bytes to store the necessary information of a trie node, as shown in the following memory layout:

```
// the layout of a hash table entry (which is also a trie node)
struct Node {
    // whether this hash table slot is occupied
    uint32_t occupyFlag:2;
    // length of this node in bytes in the trie
    uint32_t indexLen:3;
    // length of this node plus its path-compressed string
    uint32_t pcLen:3;
    // hash fingerprint, length adjustable but needs some bits
    uint32_t hash:18;
    // the minKey in this subtree
    // the first indexLen bytes is the prefix of this node in trie
    // the first pcLen bytes is the prefix + path-compress string
    uint64_t minKey;
};
```

So always spending 32 bytes to store a bitmap seems too much a waste. Instead, we will

always have slots of size 24 bytes, and "small nodes" which occupies one slot, and "large nodes" which occupies 2 slots. It is possible to have finer-grained node sizes, as discussed in paragraph 5.3, but for now we only consider two node sizes for ease of understanding and implementation.

Small nodes are nodes with no more than 8 children. We spend a byte to store the number of children in this node, and store the children as a list, each taking one byte. Large nodes are nodes with at least 8 children. The children are stored as a bitmap of 32 bytes, as described before.

The tricky thing is, how could we arrange nodes of different sizes in the same hash table? Using pointer indirection is not a solution, since we don't want to incur additional cache miss. Another good property of cuckoo hashing comes in: unlike other closed-addressing schemes that may have "clusters" of occupied slots, the possibility of each slot is occupied in cuckoo hash table is indepedent, and simply equals the load factor. So we can "steal" an unused slot from the neighborhood of the slot for the "big node" to form a 48-byte node. That unused slot should reside in the same 128-byte line, so the hardware adjacent-line prefetcher described in Section 2.3 can take advantage. Since the load factor is at most 50% and there are 4 possible slots (since a 128-byte line holds 5 slots) to steal from, with probability at least 15/16 we can find such an empty slot. If not, we fall back to the pointer-indirection solution by allocating a piece of external memory and store the pointer in the hash table. When a cuckoo replacement sequence wants to evict a "stolen slot", one can know that it is a stolen slot from its occupyFlag, and ask its owner to steal another slot instead (or fall back to pointer indirection) to make this slot free. When the "main node" itself is evicted by cuckoo replacement, it frees the slot it stole, and steals another slot (or fall back to pointer indirection) near the slot it is moved to.

Each big node of degree ≥ 8 consumes 48 bytes of memory, but the total number of "degrees" in the trie is fixed – it is always the number of nodes in the trie tree minus 1, which is $2n-1$. Since $48/8 < 24/2$, memory usage is peaked when all nodes are small nodes, and the total memory used by the data structure is bounded by 24 bytes times $2n$ nodes over 50% load factor, which is 96 bytes per 8-byte key.

Bucketized Cuckoo Hashing

We may combine the idea of adaptive node size with bucketized cuckoo hashing for even reduced memory consumption. We will have a bucket size of 4 and a hash table entry size of 16 bytes (so each bucket of entries consumes exactly one cache line). We will have 3 types of nodes: small node, medium node and large node, consuming 16, 32, 48 bytes each. As before, small node and medium node stores children directly, and will hold at most 3 / 16 children. Large node stores children in a bitmap. A node must reside in the same bucket of entries, so a large node needs to consume 3 slots in the bucket, etc.

The variant-sized nodes makes Cuckoo replacement a little different. Specifically, we may have to reserve multiple slots of space to insert a large node, and evict multiple elements in the process. This makes the theoretical analysis of Cuckoo hashing no longer working. However, since at most 1/4 of the nodes are medium nodes and at most 1/17 portion of nodes are large nodes, intuitively it should not have too much effect on the maximum load factor. Indeed, experiments show that one can still reach at least 90% load factor with various combinations of medium/large node percentage.

By averaging the size of the node over the minimum degrees, one can see that memory usage is still peaked when all nodes are small nodes. So now, the total memory usage is bounded by 16 bytes times $2n$ nodes over 90% load factor, which is around 36 bytes per 8-byte key.

However, due to time limitation, we are unable to implement this bucketized cuckoo scheme into MlpIndex . All evaluations in Section 5.4 are done using the 96-byte-per-key version.

Flat Bitmap For Top Layers of Trie Tree

This is an extra optimization for 64-bit keys. The top 3 layers of the trie tree can only hold 256^3 different nodes. It would additionally save some space, as well as DRAM accesses, to simply store the first 3 layers in a bitmap of 256^3 bits ($2MB$), instead of the hash table. After this optimization, we only need to access 6 layers times 2 (per cuckoo lookup) plus 1 (the top-layers bitmap), which is 13 DRAM positions in parallel, to execute a QueryLCP for 64-bit keys, which is another small performance and memory-consumption improvement.

The Choice of Hash Function

Finally, we need a good hash function family. The hash functions should be very efficient to compute (so it doesn't drag our performance down), vectorizable (since we need to compute multiple independent strings' hash value) and have high quality (required for cuckoo hash scheme to reach its theoretical maximum load factor). While there may definitely be other choices that meet the criteria, we chose xxHash [5], a hash function family widely used in industry that is both extremely performant and of high quality.

5.4 Evaluation

6 Conclusion

References

[1] Adam Prout. MemSQL Blog: The Story Behind MemSQL's Skiplist Indexes. Available at: https://www.memsql.com/blog/what-is-skiplist-why-skiplist-index-for-memsql.

[2] M. Ajtai. A lower bound for finding predecessors in Yao's cell probe model. Combinatorica, 8(3):235–247, Sep 1988.

[3] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, pages 521–534, New York, NY, USA, 2018. ACM.

[4] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. ACM Trans. Database Syst., 32(3), August 2007.

[5] Yann Collet. xxhash. Available at: https://github.com/Cyan4973/xxHash.

[6] Dan E.Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. https://doi.org/10.1016/0020-0190(83)90075-3.

[7] Intel. Intel Optimization Guide. Available at: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[8] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Fast: Fast architecture sensitive tree search on modern cpus and gpus. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[9] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. Cimple: Instruction and Memory Level Parallelism: A DSL for Uncovering ILP and MLP. In Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT '18, pages 30:1–30:16, New York, NY, USA, 2018. ACM.

[10] Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. Proc. VLDB Endow., 9(4):252–263, December 2015.

[11] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013), ICDE '13, pages 38–49, Washington, DC, USA, 2013. IEEE Computer Society.

[12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. EuroSys'12 - Proceedings of the EuroSys 2012 Conference, 09 2012.

[13] MemSQL. MemSQL, the no-limits database. https://www.memsql.com.

[14] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. Proc. VLDB Endow., 11(1):1–13, September 2017.

[15] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, Algorithms — ESA 2001, pages 121–133, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[16] Kenneth A. Ross. Efficient hash probes on modern processors. In 2007 IEEE 23rd International Conference on Data Engineering, pages 1297–1301, April 2007.

[17] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. Inf. Process. Lett., 6:80–82, 1977.