Practical Batch-Updatable External Hashing with Sorting

Hyeontaek Lim* David G. Andersen† Michael Kaminsky‡

Abstract

This paper presents a practical external hashing scheme that supports fast lookup (7 microseconds) for large datasets (millions to billions of items) with a small memory footprint (2.5 bits/item) and fast index construction (151 K items/s for 1-KiB key-value pairs). Our scheme combines three key techniques: (1) a new index data structure (Entropy-Coded Tries); (2) the use of sorting as the main data manipulation method; and (3) support for incremental index construction for dynamic datasets. We evaluate our scheme by building an external dictionary on flash-based drives and demonstrate our scheme's high performance, compactness, and practicality.

1 Introduction

As the amount of data stored in large-scale storage systems continues to grow, so does the importance of algorithms and data structures to provide resource-efficient access to that data. In this paper, we address the question of *external hashing*: using a small amount of RAM to efficiently locate data stored on a (large) external disk or flash drive. Our solution, *Entropy-Coded Tries*, provides three properties that are important in practice: It is memory and disk efficient; it provides fast access; and, by using sorting as its fundamental organizing method, is easy to engineer for high performance and reliability.

Traditional external hashing schemes put emphasis on the construction and evaluation speed of the hash function or the asymptotic size of the generated hash function [6, 7, 9]. However, in modern data-intensive systems, external hashing faces three challenges driven by technology and application trends:

- *Flash drives*: Flash provides fast random access speed—typically 20,000 to 1,300,000 I/Os per second (IOPS) [2, 3], compared to only a few hundred IOPS for hard drives. Flash also has fast sequential I/O, often exceeding 250 MB/s. Prior studies of external hashing only considered its use on slow hard disks; these systems, therefore, were not designed to take full advantage of high flash performance.
- *Growing cost of memory*: DRAM has become increasingly expensive and power-hungry relative to both disk and flash storage. As a result, the need for memory-efficient external hashing has grown—not only in asymptotic space use, but also in actual space use when storing millions to billions of keys.
- *Mission-critical storage systems*: Services ranging from Google and Facebook to end-user software such as web browsers use external hashing to build a highly memory-efficient external dictionary, which stores a large number of key-value pairs on disk or flash and provides fast retrieval of the data items. Ensuring the correctness and high performance of this dictionary is both critical and difficult; using a well-studied software component—sorting—as the main building block of external hashing can help achieve these goals.

The contributions of this paper are three-fold:

1. **Entropy-Coded Tries (ECT)**: ECT is a minimal perfect hashing (MPH) scheme that uses an array of highly-compressed tries on a sorted dataset in external storage. Given n distinct input keys, we assign unique values in the range $[0, n-1]$ to the input keys, whose order (or index) is determined by the hash values of the keys; we represent the mapping from the keys to their indices as tries, each of which is compressed using a new trie compression technique—a combination of Huffman coding and Elias-gamma coding that exploits the underlying statistical properties of tries created from hashed keys. ECT requires only 2.5 bits per key on average for an in-memory index (close to minimal perfect hashing's information-theoretic lower bound of approximately 1.44 bits per key [10]) and 7 microseconds to lookup a key. This CPU requirement can be reduced for very fast SSDs by trading some space efficiency.
2. **Sort as a basic building block of efficient external hashing**: We demonstrate that the use of a well-engineered sort implementation can provide high-performance indexing in ECT and even speed up other external hashing schemes that are not originally based on sorting. In particular, we develop a modified version of EPH [9] that uses sorting instead of partitioning, and show substantial performance improvements over its original partition-based scheme. ECT, which

*Carnegie Mellon University, PA, USA. hl@cs.cmu.edu
†Carnegie Mellon University, PA, USA. dga@cs.cmu.edu
‡Intel Labs, PA, USA. michael.e.kaminsky@intel.com

natively uses sorting, in turn outperforms both partitioning and sorting versions of EPH. In addition, sort provides two engineering benefits: (1) The system design becomes easy to understand and implement; and (2) the system can drop the index and fall back to binary search whenever desired. Further, a simple assessment of the system's correctness is possible by using a sort implementation that has been widely adopted in data processing systems.

3. **Incremental updates for dynamic datasets**: In workloads dealing with dynamic datasets, it is crucial to incorporate a series of updates into the existing dataset quickly. We show the benefits of supporting incremental updates in ECT as well as in our version of EPH. To handle batched updates, we do not perform the entire index construction on the pre-existent data; we partition/sort new changes only, and then sequentially merge the new changes into the previously indexed items while generating new in-memory structures during the merge. This incremental process incurs less I/O than the full construction of the dataset, which reduces update time.

2 Design

This section presents the design of Entropy-Coded Tries (ECT). It first presents a high-level overview of ECT and then describes major components of the ECT scheme in detail.

2.1 Overview ECT supports $O(1)$ retrieval of data items stored in external storage using approximately 2.5 bits of DRAM per key on average. It does so by keeping the data sorted in *hash order* (the order of the keys after applying a hash function to them), grouping adjacent keys into *virtual buckets* (each of which contains keys that share the same hash prefix of a certain length), and creating an efficient trie-based index for each virtual bucket to quickly locate the index of a particular pre-hashed key within that virtual bucket.

Figure 1 depicts the ECT construction process. Each box represents an item, and the height of the gray bar indicates the relative order of the item's hashed key. Our scheme sorts the dataset on flash and builds a trie for each disjoint group of adjacent keys (virtual bucket) in memory. During construction, the only data manipulation is sort, as virtual bucketing and trie construction do not move data on flash.

Figure 2 shows the data structures generated by the ECT scheme. A two-level index stores internal indexing information: the in-memory location of the compressed trie representation and the on-flash location of the first item of each virtual bucket. This internal index has an additional benefit in that it allows dense storage of other ECT data structures: (1) Compressed tries, one of which is generated for each virtual bucket, are stored contiguously in memory to achieve a small memory footprint; and (2) sorted hashkeys (and associated values) are kept on flash without wasting storage space.

In the following subsections, we detail the algorithms and data structures used by ECT.

2.2 Sorting in Hash Order ECT uses external sort as its sole data manipulation mechanism. Unlike many other indexing schemes, ECT does not require a specific procedure to construct a structured data layout (e.g., B-tree) or a custom data layout (e.g., EPH [9]). ECT directly applies external sort to the input dataset to obtain the final data layout for which ECT constructs an index. As sort is a well-established building block for many algorithms and systems, ECT can take advantage of readily-available sort implementations to achieve high performance and reliability [4, 15]. These sort implementations are carefully tuned for a wide range of systems (e.g., low I/O and computational demands), which greatly reduces the engineering effort needed to deliver high performance with ECT.

The unique feature of ECT with respect to sort is that it does not use the original key as the sort key. Instead, it *hashes each key* and uses this hashed result as the sort key. The choice of the hashing function is flexible as long as it provides a reasonably uniform distribution of hash values with very low or zero probability of collisions. Ideally, universal hashing [5] yields the best result for the subsequent steps, while using a conventional cryptographic hash function (e.g., SHA-1) also produces acceptable results for the workloads in our experiments.

We will refer to the hash value of each original key as a *hashkey* or simply as a key. Using hashed keys enables two important design aspects of ECT, as we explain further.

2.3 Virtual Bucketing After sorting the hashkeys,[1] ECT groups sets of adjacent keys into *virtual buckets*, partitioned by their k most significant bits (MSBs), as illustrated in Figure 1. As shown in Figure 2, the process of looking up a key involves first examining its MSBs to determine which virtual bucket it falls into. ECT consults a two-level index to find, for each virtual bucket, the location of the compressed trie index in memory, and the starting location on flash where keys for this bucket are stored.

The virtual bucketing process does not move data on storage: It merely represents a grouping for the trie indexing. Each virtual bucket's trie is compressed and decompressed independently. Compressing and reading a trie takes time roughly linear in the size of the trie, but larger tries compress

[1] In our implementation, this step occurs while sort is *emitting* the sorted keys; the process need not be complete for virtual bucketing to begin, which eliminates the need for re-reading sorted hashkeys from flash.

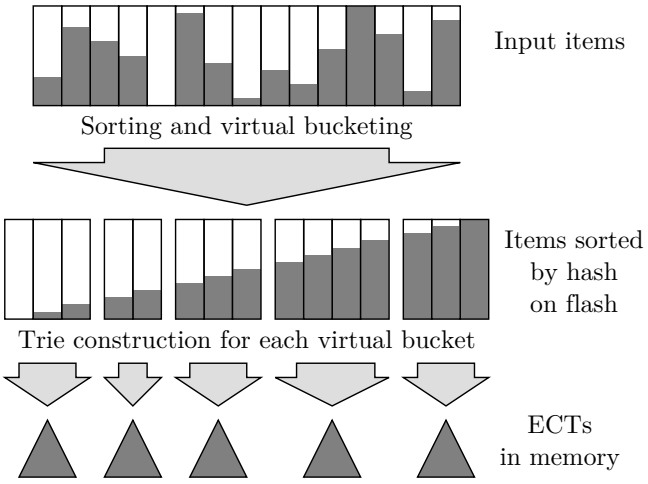

Figure 1: Workflow of the construction of ECT.

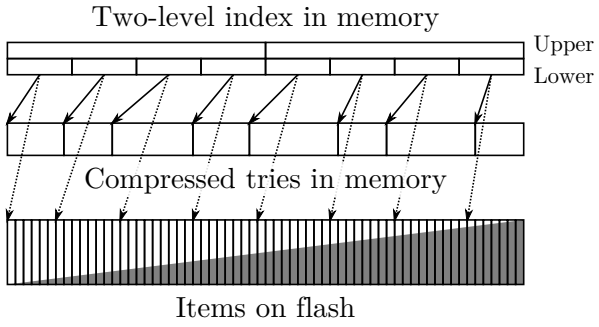

Figure 2: Constructed ECT data structures in memory and on flash.

more efficiently than small tries. As a result, the virtual bucket size g, which determines how many bits k should be used in bucketing, provides a way to trade space savings and lookup time. We show in Appendix A that the size of the largest bucket, which relates to the maximum amount of computation required for each lookup, is proportional to the average size of the buckets.

Virtual bucketing adds marginal memory overhead to index the buckets. Since both the in-memory and on-flash locations of the buckets are monotonically increasing, ECT can efficiently encode these offsets by using a two-level index. ECT records the absolute offset of every 64th bucket (when each bucket contains 256 keys on average; i.e., $g = 256$, $k = \log_2(n/256)$), and for each group of 64 buckets, it maintains the offset of individual buckets relative to the first bucket in the same bucket group. The upper level offsets use wide integers (64-bit integers in our implementation), and the lower level offsets use smaller 16-bit integers to save memory. Because of the narrow data type of the lower level, larger buckets (i.e., more than 256 keys per bucket on average) would require more upper level offsets (e.g., every 32nd bucket or more frequent) to avoid an overflow in the lower level. When the average bucket size is 256 keys, the two-level index for in-memory and on-flash locations adds insignificant space overhead of approximately 0.133 bits per key for a practical number ($< 2^{64}$) of keys.

2.4 Compressed Tries Hashkeys in each bucket are indexed by a binary trie. Similar to conventional tries, this trie represents a key with the path from the root node to a leaf node corresponding to that key, where each edge represents one bit of the key. As a result, the trie has a one-to-one correspondence between its leaf nodes and hashkeys; the i-th leaf node among all leaf nodes in the trie corresponds to the i-th smallest hashkey. A key lookup on the trie is essentially finding a leaf node corresponding to the key and counting the other leaf nodes to the left of this leaf node.

However, unlike in the tries used for text search, ECT uses *unique prefixes* of the hashkeys to remove unused information. If a leaf node does not have a sibling node, the leaf node is always removed from the trie, and its parent node becomes a new leaf node. This process significantly decreases the number of nodes in the trie while preserving the relationship between leaf nodes and hashkeys. Further, any internal node that becomes a new leaf node is an ancestor to exactly one leaf node in the original trie of full-length hashkeys. Accordingly, ECT can obtain the correct index of a lookup key by using a prefix of the key when traversing the trie: ECT can stop descending as soon as it arrives at a leaf node.

ECT reduces the amount of memory used to store the trie by applying compression. The compressed trie is represented recursively as follows:

$$Rep(T) = \begin{cases} EC(0,|T|) & \text{if } |T| = 0 \text{ or } 1, \\ EC(|Left(T)|,|T|)\ Rep(Left(T))\ Rep(Right(T)) & \\ & \text{otherwise,} \end{cases}$$

where

$$\begin{aligned} |T| &= \text{Number of keys indexed by } T \\ &\quad \text{(i.e., leaf nodes in } T\text{)}, \\ Left(T) &= \text{Left subtrie of } T, \\ Right(T) &= \text{Right subtrie of } T, \\ EC(S,C) &= \text{Entropy code for symbol } S \text{ in context of } C. \end{aligned}$$

This representation has a pre-order traversal structure similar to the recursive encoding for binary trees [11], but ours differs from the previous technique in the way it describes the left subtrie. When measuring the size of a trie,

we use the number of *keys* (i.e., leaf nodes), excluding internal nodes, of the left subtrie, and this enables an efficient entropy coding. Since the trie uses hashkeys, the keys are uniformly distributed over the entire key space; in other words, each bit of every key has the same probability of being 0 and 1. $|Left(T)|$ therefore follows the binomial distribution of $N = |T|$ and $P = 1/2$. Based on this known statistical distribution, we can encode the key counts using entropy coding. Entropy coding uses the contextual information $C = |T|$ to find a suitable code for $|Left(T)|$ because the distribution of $|Left(T)|$ depends on $|T|$. Note that for the trivial tries of $|T| \leq 1$, the entropy codes are simply an empty string, as the trivial tries have only one form (i.e., no child nodes).

When decompressing the trie representation for a lookup, C is initially set to the total number of keys in the trie. $|Left(T)|$ can be decoded from the head of the representation, and $|Right(T)|$ can be calculated by $C - |Left(T)|$, since $|Left(T)| + |Right(T)| = |T| = C$. Once $|Left(T)|$ and $|Right(T)|$ are restored, ECT recurses into the subtries by setting C to each subtrie size and using the rest of the trie representation.

In order to use the CPU cache efficienctly, we use Huffman coding only for small tries. If Huffman coding is applied to large tries, the Huffman tree or table size required for compressing and decompressing counts grows superlinearly, and this burdens the CPU cache space. Thus, we avoid using Huffman coding for a trie whose size is larger than a certain threshold. We term this threshold *hmax*; when a trie contains no more than *hmax* leaf nodes (i.e., $C \leq hmax$), we use static Huffman tables based on the binomial distributions; for larger tries, we apply Elias-gamma coding [12]. The combination of these two entropy coding techniques reduces cache misses by ensuring Huffman tables can comfortably fit in the CPU cache; we investigate the effect of this combination in Section 4.5.

To shrink the trie representation size further, we use the same code for two special cases in each trie size: $|Left(T)| = 0$ and $|Left(T)| = |T|$. That is, if a trie has all keys on either left side or right side, ECT treats them as the same because they do not affect the lookup result (recall that we only provide the correct index for a key that has been indexed, not for every possible key). This optimization reduces the compressed trie size by 0.4 bits/key.

We do not have to focus on the asymptotic space consumption of the compressed trie representations because the number of keys in a trie is practically small (e.g., around 256 keys), even when ECT indexes a large number of keys (e.g., billions of keys) across many virtual buckets. Therefore, we focus on the expected space consumption for tries in this range. Figure 3 plots the expected number of bits required to store a trie as a function of the number of keys stored in it. In this analysis, *hmax* is fixed to 64, and the result does not include the space to store the total number of keys in the trie

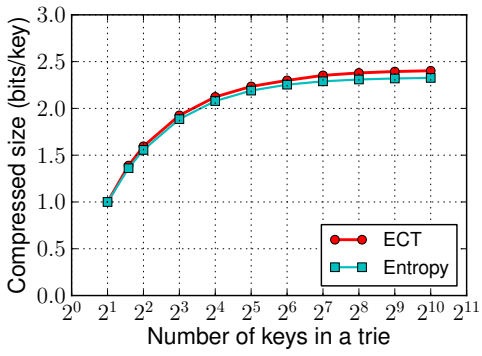

Figure 3: Expected size of the compressed representation of a single trie containing a varying number of keys.

(i.e., $|T|$ for the top-level). As the number of keys increases, the representation size approaches 2.5 bits/key. Compared to the underlying entropy of the trie, the combination of Huffman and Elias-gamma coding used in ECT achieves good coding efficiency.

2.5 Incremental Updates By employing sort as the data manipulation method, ECT can perform incremental updates efficiently, reusing the previously built sorted dataset on flash. Figure 4 illustrates the workflow of incremental updates. Only new items (not the entire dataset) are sorted and are then sequentially merged with the existing sorted items. As this sequential merge does not reorder data globally, it better uses the buffer memory for sort and thus reduces the total amount of I/O. These savings become significant as the size of the existing dataset grows larger than the size of the new dataset, as demonstrated in our evaluation (Section 4.4).

2.6 Sparse Indexing As an optimization for small items, ECT can generate more compact indexes by applying the idea of *sparse indexes*: addressing items in the disk block level, rather than the byte level. Since disk and flash drives are block-access oriented devices, they have a certain minimum I/O size (e.g., a block of 512 bytes) defined by their interface. Overheads from I/O operation processing and the physical movement of mechanical parts (e.g., disk heads) further increase the efficient minimum I/O size. Therefore, byte addresses returned by an index save no I/O compared to block addresses; by locating items in the block level, a sparse index can yield the same performance while making it more compact than byte-level dense indexes.

ECT realizes sparse indexing by pruning any subtrie that belongs to the same block. When generating the representation of a subtrie, ECT keeps track of the start and end locations of the items indexed by that subtrie. If the subtrie con-

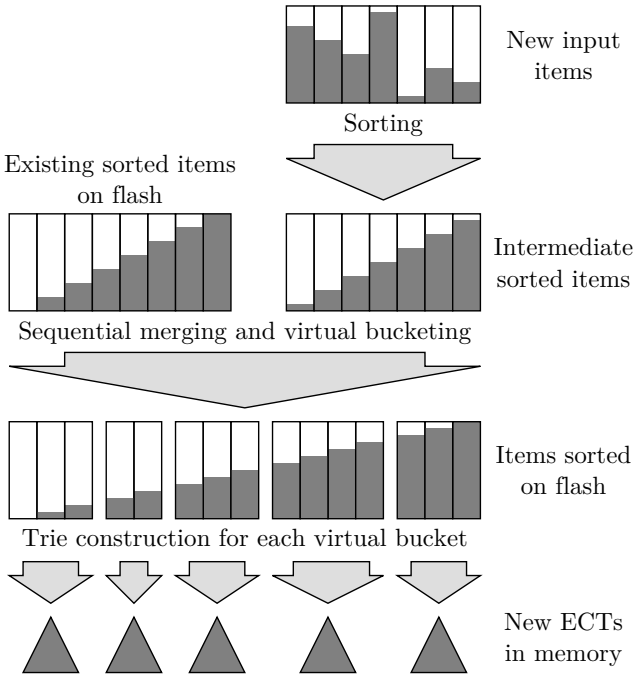

Figure 4: Incremental updates of new items to the existing dataset in ECT.

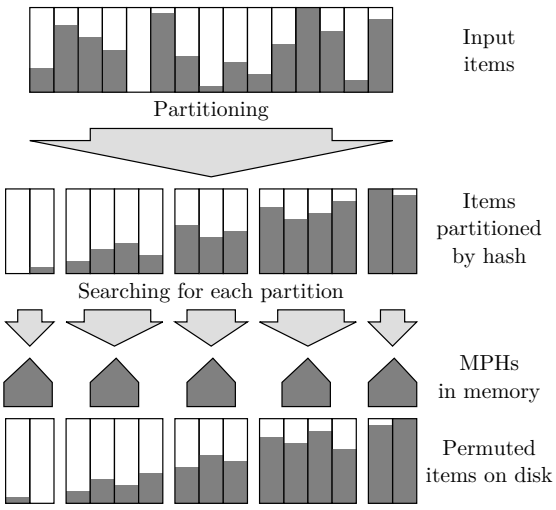

Figure 5: Comparison to the workflow of the index construction in External Perfect Hashing (EPH) [9]. Within each bucket, items are not ordered by global hash order, but permuted by a local PHF or MPHF specific to each bucket.

tains items of the same block (i.e., the start and end locations share the same block), ECT simply omits further generation of the subtrie representation. When handling a lookup, ECT repeats the same process of location tracking; when it finds a subtrie whose items fit in a single block, ECT stops decoding subtries and reads the block from the device so that it can search the queried item within the block.

Figure 6 shows expected compressed trie sizes by using sparse indexing. The trie contains 256 keys, and *hmax* is 64. With 8 or more items per block, a trie requires fewer than 1 bit per key in expectation.

The sparse indexing in ECT is stricter than k-perfect hashing [8], which allows *up to* k hash collisions. ECT generates an index with *exactly* k hash collisions (with an exception of the last block that may contain fewer than k items), and this leads to more efficient use of storage space. Further, as this sparse indexing uses the same data layout on flash, ECT can generate a new dense or sparse index on already indexed items without repeating construction process, allowing it to adapt to memory and performance requirements quickly.

3 Comparison to Other Schemes

In this section, we discuss similarities and differences between ECT and other fast external hashing schemes, summarized in Table 1.[2]

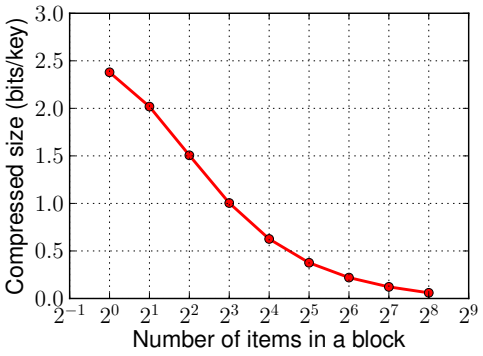

Figure 6: Expected size of the compressed representation of a single trie containing 256 items with a varying number of items in a block.

3.1 External Perfect Hashing External Perfect Hashing (EPH) [9] is a scalable external hashing scheme that requires $O(1)$ I/O operations for item retrieval and uses a small amount of memory for the index. As shown in Figure 5, EPH first partitions items into buckets using the hash values of their keys. For each bucket, EPH constructs a perfect hash

[2]The CPU requirement of MMPH is from the original MMPH paper [6].

function (PHF) or a minimal perfect hash function (MPHF), depending on the memory and storage space requirement. Since PHFs and MPHFs result in a specific offset for each key, the keys in the buckets are permuted (reordered) in their PHF/MPHF order and stored on disk. EPH achieves 3.8 bits/key of memory use, which is 27% higher than 2.5 bits/key used by ECT.

The external memory (EM) algorithm [10], which is an EPH variant that shares the same partitioning approach, still requires 3.1–3.3 bits/key as MPH to index a large number of keys that do not fit in memory.

Unlike EPH, ECT sorts all items in hash order, eliminating the additional permutation step. As we show in Section 4, EPH can benefit from being modified to use external sort, but the permutation step adds overhead as indicated in the experimental results (Section 4.2).

When performing incremental updates, EPH requires additional steps to resolve item movement within and between buckets. There is no stable ordering within a bucket across incremental updates in EPH because the order of each key is determined by an MPHF specific to the bucket, and this involves re-permutation of the items within a bucket whenever a key is added to or removed from the bucket. Worse, when the size of a bucket exceeds a certain limit or shrinks to a very small size, EPH must split or merge the buckets; as this data manipulation must be done separately from the partitioning step of EPH, support for incremental updates in EPH complicates the system design and implementation.

3.2 Monotone Minimal Perfect Hashing Monotone Minimal Perfect Hashing (MMPH) [6, 7, 13], like ECT, uses sorting as its data manipulation method, and builds its index on a sorted table. While ECT uses hash order for sorting and trie construction, MMPH uses original keys, and thus, requires a variable amount of computation and memory space depending on the key distribution and characteristics. Among various low-level indexing methods for MMPH, "hollow tries" provides the smallest hash function size—4.4 bits/key—for random keys [6], which is the best workload for minimizing the index size.

Although MMPH shares several characteristics with ECT in that both use sort, we focus our evaluation on comparisons between ECT and EPH because both yield smaller index sizes than MMPH.

4 Evaluation

This section presents the performance evaluation of an external dictionary using ECT as its index. As a comparison, we modified the original EPH implementation, which only supported a hash function interface, to provide the full functionality of an external dictionary. Our modified version of EPH can operate using its custom partitioning as well as using external sort similarly to ECT, as sort results are compatible with partition results. In addition, we implemented optional incremental construction in ECT and extended EPH to support the same incremental construction functionality, because the original EPH implementation only supported index construction from scratch.

Component	Specification
CPU	Intel Core i7 860 (quad-core) @ 2.80 GHz
DRAM	DDR3 4 GiB
SSD	RAID-0 array of three Intel SSD 510 120 GB (MLC)

Table 2: Experimental system setup.

Throughout the evaluation, we explicitly indicate binary prefixes (powers of 2) using "i" (e.g., MiB, GiB) to avoid confusion with SI prefixes (powers of 10) (e.g., MB, GB).

4.1 Experiment Setup Table 2 shows the hardware configuration for the experiments. All input, output, and temporary data are stored on a RAID-0 array consisting of three SATA SSDs, each of which provides up to 265 MB/s sequential read and 200 MB/s sequential write throughput.

We use the `WEBSPAM-UK2007` URL list [1] as input keys (on average 111 bytes per URL) and associated values of 1000 bytes with each key.[3] Due to the large size of the key-value pairs, we use up to 24 million unique URLs at the beginning of the URL list; as we use hashkeys, using the first part of the URL list does not introduce skew in the input data. Unless specified, this input is the standard workload in this section.

In addition, to examine the performance with a large number of items, we use 1 billion small key-value pairs, which consist of 64-byte keys and 4-byte values. Each key is hashed into 12 bytes at the beginning of the index construction, constituting a 16-byte key-value pair on flash during and after construction to support later key-value retrieval. While we could use the sparse indexing technique (Section 2.6), we avoid using it for ECT to make a fair comparison with EPH, which does not have an implementation for the technique. This workload is denoted as `small-item`, and its experiment result is shown in Section 4.6.

All experiments were performed on Ubuntu 11.04 (64-bit), and we used Nsort [15] as an external sort implementation. Each configuration of the experiments involving time measurements had three runs, and the range of the results is indicated with an error bar.

[3] Note that using a 1000-byte value for each key significantly increases both data size and construction cost compared to prior studies [6, 9], which stored no key (and value) data in external storage; our study stores both key and value data on flash to provide a dictionary interface (retrieval of a value associated with a key).

	External Perfect Hashing (MPH version)	Monotone Minimal Perfect Hashing (MMPH)	Entropy-Coded Tries (ECT)
Data manipulation method	Partitioning with hash + permutation by MPHFs	Sorting in original key order	Sorting in hash order
Low-level index type	MPH	Various	Compressed trie
I/O complexity for construction	$O(n)$	$O(n)$	$O(n)$
Memory requirement for a large set of keys	3.8 bits/key	$\geq$ 4.4 bits/key	2.5 bits/key
CPU requirement for an in-memory lookup	0.6 µs/lookup	7 µs/lookup	7 µs/lookup

Table 1: Summary of the comparison between ECT and other fast external hashing schemes.

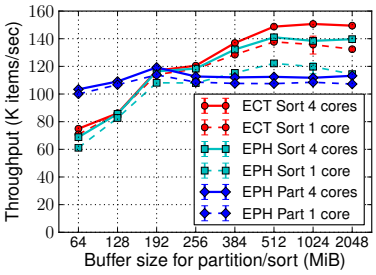

Figure 7: Construction performance with varying buffer size for partition/sort.

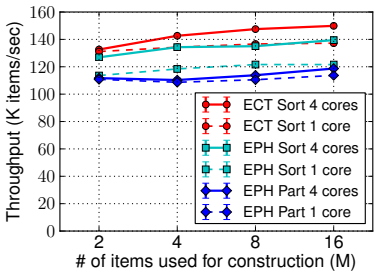

Figure 8: Construction performance with a varying number of items.

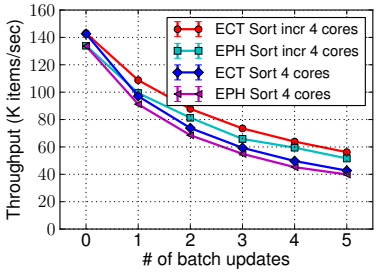

Figure 9: Construction performance for different combinations of indexing and incremental construction schemes. Each batch contains 4 M new items.

4.2 Construction Initial construction of the external hashing involves a large amount of I/O for input data, temporary files, and output data. I/O is a major bottleneck for the overall construction process. Both Nsort and EPH provide a settable parameter that controls the size of in-memory buffer space to use I/O efficiently for sorting and partitioning; thus, determining the appropriate buffer size is important. An excessively large buffer may decrease overall performance because the OS page cache, which competes for the same main memory space, is also important for reading the input data and writing the final output.

Figure 7 shows the construction performance when using different buffer sizes for the partition and sort stage. This experiment used 16 million key-value pairs, whose size (about 16 GB) far exceeds the total memory size (4 GiB). For both ECT and EPH, the sort versions that use only 1 core (dotted lines) achieve the best performance using 512 MiB for buffering, while the partition-based EPH (EPH Part) performs best with a smaller buffer size of 192 MiB (Table 3). This difference is because Nsort internally uses direct I/O that bypasses the OS page cache, and thus its performance is less susceptible to the reduced amount of the OS page cache when the in-memory sort buffer size is large. With 4 cores (solid lines), ECT works best with 1024 MiB buffer size. From this result, we can observe major performance boosts when using multiple cores due to Nsort's internal optimization, but not with EPH's custom partitioning, because EPH's custom data partitioning code uses 1 core and does not directly benefit from multiple cores. This demonstrates the performance and engineering advantages of using a well-understood and well-optimized primitive such as sorting for the main data manipulation.

In the subsequent experiments, we used the best buffer memory size for each configuration. Figure 8 plots the construction performance with varying dataset size; we set the buffer size based on the large dataset of 16 M items, which shows that smaller dataset sizes have an insignificant effect on the construction speed.

4.3 Index Size Despite its faster construction speed, ECT generates a smaller index than EPH. To index 16 million items of the given dataset, ECT used 5.02 MB (2.51 bits/key); EPH required 7.83 MB (3.92 bits/key) excluding a fixed-size lookup table of 3 MiB, or 10.98 MB including the lookup table (5.49 bits/key in total). ECT's space consumption is at least 36% lower than EPH's and is close to the analytical result shown in Figure 3.

4.4 Incremental Updates For datasets that are not fully static, incremental updates improve the reconstruction speed

	Throughput @ Buffer size
ECT Sort 4 cores	151 K items/s @ 1024 MiB
ECT Sort 1 core	138 K items/s @ 512 MiB
EPH Sort 4 cores	141 K items/s @ 512 MiB
EPH Sort 1 core	122 K items/s @ 512 MiB
EPH Part 4 cores	119 K items/s @ 192 MiB
EPH Part 1 core	114 K items/s @ 192 MiB

Table 3: Best construction performance of each scheme.

	Throughput	Total CPU time
EPH	69.16 K queries/s	139.76 s
ECT	64.02 K queries/s	217.75 s
(difference)	-7.4%	+55.8%

Table 4: Random lookup performance with random queries using 16 threads.

of the external hashing substantially. Figure 9 plots performance boosts with incremental updates. The workload constructs an index using 4 M items. Then, it adds another 4 M items on each batch to update the existing dataset. It repeats this update up to 5 times until the final dataset size reaches 24 M items. As clearly shown, using incremental updates outperforms versions that must rebuild the sorted/partitioned dataset from scratch rather than reusing the previously organized data. The performance gap increases as the number of updates increases—more than 20% with 5 updates. Therefore, allowing incremental updates is important to improving the performance of storage systems with dynamic data.

4.5 Space vs. Lookup Speed Tradeoff Table 4 shows the full-system lookup performance difference between ECT and EPH, using 8 M random queries on 16 M items. Each lookup includes data item retrieval from flash, thus incurring I/O. The experiment used 16 threads to take advantage of the I/O parallelism of flash drives [14]. While both algorithms exhibit lookup speed exceeding 64 K queries/s, ECT is 7.4% slower than EPH. This difference mostly comes from the higher cost of ECT's lookup process, as shown in the total CPU time, which also includes the CPU overhead of I/O processing in the kernel. However, the difference in the actual lookup speed is much smaller than the difference in the total CPU time, since the system's CPU is largely underutilized (i.e., external lookup is an I/O-bound task). Therefore, for a small extra end-to-end lookup cost, ECT brings great savings in the index size (36% smaller index size than EPH). The amount of I/O performed was the same in both schemes, as they incur a single random I/O per lookup.

When a bigger and faster array of flash drives is used, slow hash function evaluation may cause the underutilization of the storage array. ECT provides two knobs—the virtual bucket size (g) and the maximum trie size at which to apply

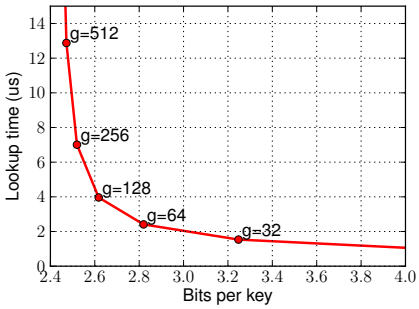

Figure 10: The tradeoff between size and in-memory lookup performance on a single core when varying average bucket size (g) with $hmax = 64$.

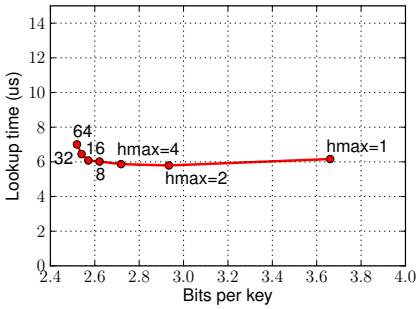

Figure 11: The tradeoff between size and in-memory lookup performance on a single core when varying maximum trie size for Huffman coding ($hmax$) with $g = 256$.

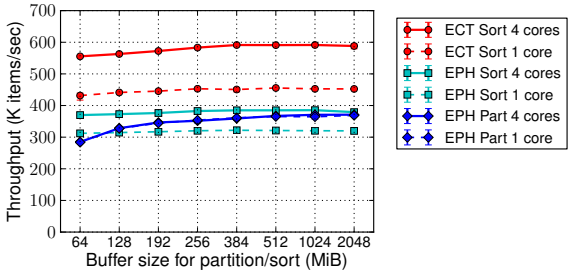

Figure 12: Construction performance with varying buffer size for partition/sort (`small-item`).

	Throughput @ Buffer size
ECT Sort 4 cores	591 K items/s @ 1024 MiB
ECT Sort 1 core	455 K items/s @ 512 MiB
EPH Sort 4 cores	385 K items/s @ 1024 MiB
EPH Sort 1 core	322 K items/s @ 384 MiB
EPH Part 4 cores	371 K items/s @ 2048 MiB
EPH Part 1 core	369 K items/s @ 2048 MiB

Table 5: Best construction performance of each scheme (`small-item`).

Huffman coding (*hmax*)—to allow the system to obtain a balance between the in-memory data structure size and per-lookup computation. To highlight the effect of adjusting knobs, we measure in-memory (no I/O) lookup performance using a single core only.

Figure 10 shows that by setting the average virtual bucket size (g) to 256 keys, ECT requires as little as 2.5 bits per item and CPU consumption for an in-memory lookup of 7 µs. This lookup speed allows the system to support up to 67 K queries per second per core without considering I/O costs. With such low memory use by ECT, if a system has 4 GiB of DRAM to store the ECT index, it can index 13.7 billion items. By having a smaller number of keys in each virtual bucket, the system can trade the memory efficiency for even higher lookup speed.

Figure 11 shows diminishing returns of using a high value for the maximum trie size to apply Huffman coding (*hmax*). With a moderate *hmax* value (8 to 64), the system can achieve small index size as well as low lookup time.

4.6 Small Items With small items of 64-byte key-value pairs, the construction process is more dependent on computation than with large items. As shown in Figure 12 and Table 5, ECT using 4 cores exceeds any EPH scheme's performance by far; ECT's simple construction process handles 591 K key-value items per second (28 minutes in total), achieving 54% higher construction speed than the best EPH scheme.

ECT demonstrated its memory efficiency again with small items. ECT required 35% less memory space than EPH; ECT's index size was 314 MB (2.51 bits/key), whereas EPH required 481 MB (3.85 bits/key) when excluding a fixed-size lookup table of 3 MiB, or 484 MB with the lookup table (3.87 bits/key in total).

5 Conclusion

Entropy-Coded Tries (ECT) provide a practical external hashing scheme by using a new trie-based indexing technique on hash-sorted items. Our trie compression yields a very compact hash function occupying 2.5 bits per key, with fast lookup time (around 7 µs), making ECT suitable for datasets stored on high-speed storage devices such as flash drives. Using sort as the main data manipulation method substantially saves engineering effort, and this enables ECT to use I/O and multi-core CPUs efficiently for index construction and incremental index updates.

Acknowledgments

This work is supported by National Science Foundation award CCF-0964474, Google, Intel Science and Technology Center for Cloud Computing, and the Parallel Data Lab member companies. Hyeontaek Lim is supported by the Facebook Fellowship. We would like to thank Guy Blelloch, Danny Sleator, Michelle Mazurek, and anonymous reviewers of ALENEX13 for their valuable feedback and Fabiano C. Botelho for providing the source code of EPH.

References

[1] WEBSPAM-UK2007. http://barcelona.research.yahoo.net/webspam/datasets/uk2007/links/, 2007.

[2] Fusion-io ioDrive Octal. https://www.fusionio.com/platforms/iodrive-octal/, 2012.

[3] Intel Solid-State Drive 510 Series. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-510-series.html, 2012.

[4] Sort benchmark home page. http://sortbenchmark.org/, 2012.

[5] N. Alon, M. Dietzfelbinger, P. Miltersen, E. Petrank, and G. Tardos. Linear hash functions. *Journal of the ACM (JACM)*, 46(5):667–683, 1999.

[6] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practise of monotone minimal perfect hashing. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments*, ALENEX '09. Society for Industrial and Applied Mathematics, 2009.

[7] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with O(1) accesses. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 785–794. Society for Industrial and Applied Mathematics, 2009.

[8] D. Belazzougui, F. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the 17th European Symposium on Algorithms*, ESA '09, pages 682–693, 2009.

[9] F. C. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 653–662. ACM, 2007.

[10] F. C. Botelho, R. Pagh, and N. Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 2013. To appear.

[11] D. R. Clark. *Compact PAT trees*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1998.

[12] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, Mar. 1975.

[13] R. Grossi and G. Ottaviano. Fast compressed tries through path decompositions. In *Proceedings of the Annual Meeting on Algorithm Engineering and Experiments*, ALENEX '12. Society for Industrial and Applied Mathematics, 2012.

[14] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[15] C. Nyberg and C. Koester. Ordinal Technology - Nsort home page. http://www.ordinal.com, 2012.

[16] M. Raab and A. Steger. "Balls into bins" — a simple and tight analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '98, pages 159–170, London, UK, 1998. Springer-Verlag.

Appendices

A Maximum Size of Virtual Buckets

The size of the largest virtual buckets is $\Theta(n/m)$, where n is the total number of keys indexed by ECT, and m is the number of virtual buckets ($m = 2^k$ when using the k MSBs to determine the virtual bucket of a key). We can formulate the problem using a conventional balls-into-bins model by treating each key as a ball (total n balls) and each virtual bucket as a bin (total m bins). It is known that when $n \geq m \log_2 m$, the maximum number of balls in each bin is $\Theta(n/m)$ [5, 16]. In our setting, let m be n/w (each bin contains w balls on average), where w is the width of a machine word (e.g., 64)—this will require using $\log_2(n/w)$ MSBs for determining to which virtual bucket each key belongs. Since $n/w = 2^{\log_2 n - \log_2 w}$, $m \log_2 m = (n/w)\log_2(n/w) = 2^{\log_2 n - \log_2 w} \cdot (\log_2 n - \log_2 w) = 2^{\log_2 n - \log_2 w + \log_2(\log_2 n - \log_2 w)}$. Observe $n \leq 2^w$ (due to addressing items by a machine word), $2^{\log_2 n - \log_2 w + \log_2(\log_2 n - \log_2 w)} \leq 2^{\log_2 n - \log_2 w + \log_2(w - \log_2 w)} < 2^{\log_2 n} = n$. Thus, $n \geq (n/w)\log_2(n/w)$, and we get the maximum load of each virtual bucket of $\Theta(n/(n/w)) = \Theta(w)$. In other words, as long as the average bucket size ($g = n/m$) is no smaller than the number of bits in a machine word, the size of the largest bucket will remain proportional to the average size of all buckets.