Memory-Efficient GroupBy-Aggregate using Compressed Buffer Trees

Hrishikesh Amur[†], Wolfgang Richter[⋆], David G. Andersen[⋆],
Michael Kaminsky[‡], Karsten Schwan[†], Athula Balachandran[⋆], Erik Zawadzki[⋆]

[⋆]*Carnegie Mellon University,* [†]*Georgia Institute of Technology,* [‡]*Intel Labs*

Abstract

The rapid growth of fast analytics systems, that require data processing in memory, makes memory capacity an increasingly-precious resource. This paper introduces a new compressed data structure called a *Compressed Buffer Tree* (CBT). Using a combination of techniques including buffering, compression, and serialization, CBTs improve the memory efficiency and performance of the GroupBy-Aggregate abstraction that forms the basis of not only batch-processing models like MapReduce, but recent fast analytics systems too. For streaming workloads, aggregation using the CBT uses 21-42% less memory than using Google SparseHash with up to 16% better throughput. The CBT is also compared to batch-mode aggregators in MapReduce runtimes such as Phoenix++ and Metis and consumes 4× and 5× less memory with 1.5-2× and 3-4× more performance respectively.

1 Introduction

This paper presents the design, implementation, and evaluation of a new data structure called the *Compressed Buffer Tree* (CBT). The CBT implements in-memory GroupBy-Aggregate: Given a set of records, partition the records into groups according to some key, and execute a user-defined aggregation function on each group. The CBT uses less memory and provides more performance than current alternatives.

SoCC'13, Oct. 01–03 2013, Santa Clara, CA, USA.
ACM 978-1-4503-2428-1/13/10.
http://dx.doi.org/10.1145/2523616.2523625

Instance type (size)	Percentage of hourly cost		
	CPU	Memory	Storage
Std. (S)	18%	**47%**	35%
Std. (L)	16%	**44%**	40%
Hi-Mem. (XL)	17%	**69%**	14%
Hi-CPU (M)	**40%**	20%	**40%**

Table 1: Amazon EC2 resource costs (# resource units × per-hour unit resource cost); details in Section 5.2.1.

The CBT focuses on *in-memory* GroupBy-Aggregate (called *aggregation* henceforth) because of the recent need for performing aggregation with not just high throughput, but low latency as well. This trend is driven by *fast analytics* systems such as Muppet [32], Storm [3] and Spark [49], that seek to provide real-time estimates of aggregate data such as customer checkins at various retailers or trending topics in tweets. For these systems, the conventional model of batch-aggregating disk-based data does not yield the necessary interactive performance. Instead, fast and incremental aggregation on summarized data in memory is required.

The CBT focuses on performing *memory-efficient* aggregation because recent hardware trends indicate that memory capacity is growing slowly relative to compute capacity [35]. Moreover, our analysis in Table 1 of prices charged by Amazon for different resources in EC2 instances shows that for most instances, memory costs already dominate, in terms of the proportion of total instance cost. In other words, with Amazon's standard configurations, it is far cheaper to double the CPU capacity than to double the memory capacity. This trend, seen in the context of heavy demand for memory capacity for fast analytics, naturally motivates the development of techniques that can trade off additional CPU use in return for reduced memory consumption.

Memory compression is used by the CBT to exploit this trend, as in previous systems [5, 43]. The CBT efficiently maintains intermediate aggregate data in compressed form, trading off extra compute work for reduced

memory capacity use. Direct application of memory compression to existing methods for in-memory aggregation does not work. Consider hashing, a popular technique for fast in-memory aggregation [12, 32]: To maintain, for example, a running count of real-time events, each new event is hashed using its unique key to a hashtable bucket and the stored counter is incremented. Compressing buckets of the hashtable in memory is ineffective; compressing individual buckets makes compression inefficient as each bucket may be small, and compressing many buckets together as a block causes frequent decompression and compression whenever a bucket in the block is accessed. Hashtables, therefore, are not ideal for aggregating compressed data.

The CBT solves this problem with the insight that accessing compressed data is similar to accessing data on disk. In both cases, (a) there is a high static cost associated with access (i.e., decompression or reading from disk), and (b) an access returns more data than requested (i.e. entire compressed blocks or disk pages). The well-known External Memory model [6] is developed around precisely these constraints. Techniques based on this model, exploit these constraints by organizing data on disk such that all the data in the block that is returned from an access is "useful" data. This allows the static access cost to be amortized across all the data in the block. Most importantly, techniques developed for this model are analogously applicable to compressed memory.

In particular, the CBT leverages the *buffer tree* data structure [7] as the store for compressed aggregate data. The CBT intelligently organizes the aggregate data into compressed buffers in memory such that data that is compressed together also tends to be accessed together. This allows the cost of decompression and compression to be amortized across all the data in the buffer. High compression efficiency is maintained by always compressing *large* buffers. This allows aggregation to be both extremely fast and memory-efficient.

Concretely, our contributions in the paper are:

- We introduce the Compressed Buffer Tree (CBT), a data structure for stand-alone, memory-efficient aggregation written in C++; the CBT seeks to reduce memory consumption through compression. To reduce the number of compression/decompression operations (for high performance), the CBT leverages the buffer tree data structure. The novelty of this work lies in two insights: (a) the buffer tree, originally designed to minimize I/O, can also be used to reduce the number of compression/decompression operations, and (b) the buffer tree can be used to support the GroupBy-Aggregate abstraction, hitherto only implemented using sorting or hashing [48, 22]. This abstraction is a building block for many data-processing models.
- The CBT includes optimizations to specifically improve aggregation performance (e.g. multi-buffering, pipelined execution) and reduce memory (e.g. serialization, column-wise compression), which were not the focus of the original buffer tree work.
- For stream-based workloads prevalent in fast analytics, the CBT uses 21-42% less memory than an aggregator based on Google's `sparsehash` [21], a memory-efficient hashtable implementation, with up to 16% higher aggregation throughput. The CBT can also be used as a batch-mode aggregator. Compared to state-of-the-art batch-mode aggregators from the Phoenix++ [42] and Metis [37] MapReduce runtimes, the CBT uses 4× and 5× less memory, achieving 1.5-2× and 3-4× higher throughput, respectively. Our contribution can be viewed as either allowing existing aggregation workloads to use less memory, or accommodating larger workloads on the same hardware.

We organize the paper as follows: The rationale behind the choice of the Buffer Tree is explained in Section 2. CBT design is detailed in Section 3 and implementation details and various optimizations are described in Section 4. Aggregator comparisons and an in-depth CBT factor analysis are presented in Section 5.

2 Aggregator Data Structures

In this section, we first introduce a well-known model for external memory; we then discuss the applicability of the model to compressed memory. After a brief survey of design alternatives, we explain why we choose the buffer tree as the basis for a memory-efficient aggregator.

2.1 The External Memory (EM) Model

The External Memory model [6] is a popular framework for the comparison of external data structures and algorithms. It models the memory hierarchy as consisting of two levels (e.g. DRAM and disk). The CPU is connected to the first level (sized M) which is connected, in turn, to a large second level. Both levels are organized in blocks of size B and, transfers between levels happen in units of blocks. Note that the model can be used to describe various combinations of levels: hardware last-level cache (LLC)/DRAM or DRAM/hard disk etc. In this section, we will use the (capitalized) terms *Cache* and *Disk* to indicate the two levels, and the terms *Read* and *Write* to indicate the movement of a block of data from *Disk* to *Cache* and *Cache* to *Disk* respectively.

Term	DRAM/ Hard disk	Uncompressed/ Compressed DRAM
Cache	DRAM	DRAM
Disk	Hard drive	Compressed DRAM
Read	Page read	Block decompress
Write	Page write	Block compress
I/O	Page read/write	Block compress/ decompress

Table 2: External Memory (EM) Model analogs for compressed memory

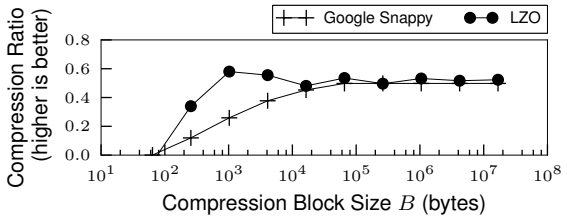

Figure 1: Efficient compression requires large block sizes. Dataset used: Snapshot of Wikipedia compressed using Snappy [20] and LZO [2] compression libraries.

2.2 EM Model for Compressed Memory

Our goal is to improve the memory efficiency of GroupBy-Aggregate by maintaining aggregate data in compressed form. Data access necessitates decompression, so the organization of data plays a critical role in determining the frequency of compression, which affects overall aggregation performance. Our first contribution is to study techniques for this organization of data in the EM model. More precisely, compressed memory is modeled as *Disk* and uncompressed memory as *Cache*. Therefore, a *Read* is a decompression operation and a *Write* is a compression operation. Table 2 shows the analogs for uncompressed/compressed memory compared to the conventional DRAM/disk hierarchy.

The EM model requires that data movement between *Disk* and *Cache* are in blocks of size B. For compressed memory, Figure 1 shows that compression efficiency is high only when moderately large blocks are compressed together. This holds true for many popular compression algorithms based on Lempel-Ziv techniques [50, 51], because these work by building a dictionary of frequently-occurring strings and encoding the strings with (the much shorter) references to the dictionary. Only large blocks allow long, repeating runs to be captured.

2.3 Data Structure Alternatives

Next, we discuss alternatives for a data structure for storing data on *Disk*. We consider data structures that support

Data Structure	*Reads* per `get`	*I/O*s per `insert`
B-Tree	$O(\log_B N)$	$O(\log_B N)$
Hashing [17]	$O(1)$	$O(1)$
LSM Tree [41]	$O(\log N \log_B N)$	$O(\frac{1}{B}\log N)^a$
COLA [8]	$O(\log^2 N)$	$O(\frac{1}{B}\log N)^a$
Buffer tree [7]	$O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})^a$	$O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})^a$
Append-to-file	$O(\frac{N}{B})$	$O(\frac{1}{B})^a$

Increasing `insert` performance

[a] amortized

Table 3: Comparison of I/Os performed by various data structures to `insert` a new key-value pair or `get` the value associated with a key in the External Memory model: B is the block size for data movement between *Disk* and *Cache* (see §2.1), M is the size of the *Cache* and N is the number of unique keys.

two operations: 1) `insert(key, value)` – insert the key, if absent, and associate the value with the key, and 2) `get(key)` – fetch the current value associated with the key. The performance of `insert` and `get` is measured in terms of *I/O*s required per operation. Previous work offers data structures that provide a spectrum of tradeoffs between `insert` and `get` performance under the EM model. Table 3 summarizes these in increasing order of `insert` performance.

At one end of the spectrum are B-trees and hashing [17]. Both methods provide low latency `get`s, but modify data on *Disk* in-place. This means that each `insert` operation requires one *Read* and one *Write*. At the other end are unordered log-structured stores (append-to-file in Table 3) such as the Log-Structured File System [44]. They are called *unordered* because they do not order keys in any way (e.g. sort) before *Writing* to *Disk*. Typically, a B-sized buffer in *Cache* is maintained and new key-value pairs copied into it. When full, it is *Written* to a log on *Disk*. This amortizes the cost of the *Write* across all key-value pairs in the block providing the optimal cost of $O(\frac{1}{B})$ *Write*s per `insert`. However, `get`s need to *Read* the entire log in the worst case.

Ordered log-structured stores form the middle of the spectrum. These buffer multiple key-value pairs in the *Cache* and *order* them before writing to *Disk*. Ordered log-structured stores offer much higher `insert` performance compared to in-place update structures like B-trees, and also provide the crucial advantage that key-value pairs with the same key tend to cluster together in the data structure allowing periodic merge operations to be performed efficiently. Examples of ordered log-structured stores include Log-Structured Merge (LSM) Trees [41], Cache-Oblivious Lookahead Arrays (COLA) [8] and buffer trees [7]. Similar to unordered

stores, ordered stores amortize the cost of *Writes* across multiple key-value pair aggregations, but, unlike them, periodically merge records belonging to the same key.

LSM Trees maintain multiple B-Trees organized in levels with exponentially increasing size. When a tree at one level becomes full, it flushes to the tree at the next level. COLA are similar to LSM Trees, but use arrays instead of B-trees at each level for slightly slower reads. Both LSM Trees and COLA, with their emphasis on maintaining low latency for *Reads*, aim to serve as online data structures and are used as data stores in key-value databases [11]. The buffer tree has a slightly different goal. It targets applications that *do not* require low latency queries, and only provides good amortized query performance, as shown in Table 3. In return, buffer trees make better use of available *Cache* space, to improve on the `insert` performance of LSM Trees and COLA.

Briefly, the buffer tree maintains an (a,b)-tree with each node augmented by a nearly *Cache*-sized buffer. Inserts and queries are simply copied into the root buffer. When any node fills up, it is emptied by pushing the contents of the node down to the next level of the tree. During the emptying process, multiple `insert`s with the same key can be coalesced.

We leave proofs regarding the performance bounds to related work. Having outlined the tradeoffs in `insert` and `get` performance, we look at what properties are required from a data structure to support GroupBy-Aggregate efficiently.

2.4 GroupBy-Aggregate in the EM Model

To understand the requirements of GroupBy-Aggregate, consider implementing GroupBy-Aggregate using a hashtable on *Disk* to store aggregate data. Recall that contents of the hashtable on *Disk* can only be accessed in blocks of size B. To aggregate a new key-value pair (k,v), the current value V_0 associated with k (if any) must be fetched using a `get`; the value v is then aggregated with V_0 to produce V_1, and (k,V_1) `insert`ed. Thus, each aggregate requires a `get` and an `insert`. The problem is that if each key-value pair is much smaller than a block, the remaining data in the block that has been read is wasted. We term this type of *read-modify-update* aggregation as *eager* aggregation, since each new key-value is immediately aggregated into the current value.

An alternative is *lazy* aggregation, where up-to-date aggregates of all keys are not always maintained. Eager aggregation assumes the worst: that aggregated values for any keys may be required at any point during aggregation. In lazy aggregation, on the other hand, for a new key-value pair (k,v) that has to be aggregated, aggregation is simply *scheduled* by `insert`ing the pair. Of course, this requires a lazy aggregator to be *finalized*, where all scheduled-but-not-executed aggregations are performed, before the aggregated values can be accessed.

Lazy aggregation works well in scenarios where the accessing of aggregated values happens at well-defined intervals (e.g. MapReduce and databases where the aggregated results are only accessed at the end of the job and query respectively), or when the ratio of aggregations to accesses is high. For example, lazy aggregation works well for the following scenario: Find the top-100 retweeted tweets, updated every 30 seconds, by monitoring a stream of tweets; the incoming rate of all tweets is high, possibly millions per second, and the aggregated data which includes the top-100 tweets is only accessed once (therefore requiring a finalize) every 30 seconds.

To contrast the requirements of a lazy aggregator with an eager aggregator, it is immediately clear that, for the former, the `get` corresponding to accessing the current aggregated value can be avoided. Therefore, a data structure used to support a lazy aggregator only needs to support fast `insert`s. There is one other requirement: For fast finalization, the data structure *must* be ordered. To see why, consider the case of append-to-file from §2.3. Append-to-file (unordered log) supports very fast `insert`s, but at finalization time, the `insert`ed values for a given key can be distributed over the entire log on *Disk*. To collect these, the entire log essentially has to be sorted, which leads to poor finalization performance. Instead, an ordered data structure clusters `insert`ed values with the same key together, allowing periodic partial aggregation. This improves finalization performance. For example, for LSM and buffer trees, buffered values in the higher levels must only be flushed down to the lowest level during finalization, which is significantly faster than sorting the entire log.

Taking these requirements into account, along with conclusions from §2.3, it is clear that buffer trees provide the solution we seek: low-cost `insert`s for fast scheduling of aggregation, and ordering for efficient finalization. Using this insight, our data structure, the *Compressed Buffer Tree* (CBT), uses the buffer tree to organize aggregate data in compressed memory.

3 Compressed Buffer Trees

The CBT is a data structure for maintaining key-value pairs in compressed memory. It allows new key-value pairs to be inserted for aggregation; it merges inserted key-value pairs with existing key-value pairs in compressed memory while reducing the number of compression/decompression operations required. We adopt the CBT for use as a fast, memory-efficient aggregator of key-value pairs that can be used to implement a generic GroupBy-Aggregate operation. This aggregator can be

used in a variety of data-processing runtimes including MapReduce and fast analytics systems.

3.1 Partial Aggregation Objects (PAOs)

The CBT uses a simple data structure called a Partial Aggregation Object (PAO) to represent intermediate partial results. Yu et al. [48] hint at a similar abstraction, but we make it explicit to help simplify the description of CBTs.

Prior to aggregation, each new key-value pair is represented using a PAO. During aggregation, PAOs accumulate partial results. PAOs also provide sufficient description of a partial aggregation such that two PAOs p^q and p^r with the same key can be *merged* to form a new PAO, p^s. Because different PAOs with the same key provide no guarantees on the order of aggregation, applications must have a `merge` function that is both commutative and associative. For example, in a word-counting application, two PAOs: $\langle$`arge`$, 2, f :$ `count()`$\rangle$ and $\langle$`arge`$, 5, f :$ `count()`$\rangle$, can be merged because they share the same key "`arge`", invoking the function associated with the PAOs `count()`, and resulting in $\langle$`arge`$, 7, f :$ `count`$\rangle$. The CBT is programmed by the user by specifying the structure of the PAO (members and function).

3.2 The CBT API

The CBT API consists of two mutating operations: `insert(PAO p)` – schedule a PAO for aggregation, and `finalize()` – produce final aggregate values for all keys. Aggregated PAOs can then be read using an iterator, but random key lookups are not currently supported (we briefly address this in Section 7).

3.3 CBT Operation

Like the buffer tree, the CBT uses an (a,b)-tree [25] with each node augmented by a large memory buffer. An (a,b)-tree has all of its leaf nodes at the same depth and all internal nodes have between a and b children, where $2a \leq b$. The entire CBT resides in memory. We term all nodes except the root and leaf nodes "internal nodes." The root is uncompressed, and the buffers of all other nodes are compressed.

Inserting PAOs into the CBT When a PAO is `inserted` into the CBT, the tuple $\langle$`hash, size, serialized PAO`$\rangle$ is appended to the root node's buffer; `hash` is a hash of the key, and `size` is the size of the serialized PAO. We use the hash value of the key, instead of the key itself, because string comparisons are expensive (excessively so for long keys). Handling hash collisions is explained later. PAOs have to be serialized into the root buffer because PAOs, in general, can contain pointers, dynamic data structures such as C++ vectors etc., and because the eventual goal is to compress the buffer, the PAOs have to serialized into a contiguous chunk of memory. PAOs are copied into the root node until it becomes full. The procedure to empty a full node (Algorithm 1) is summarized next.

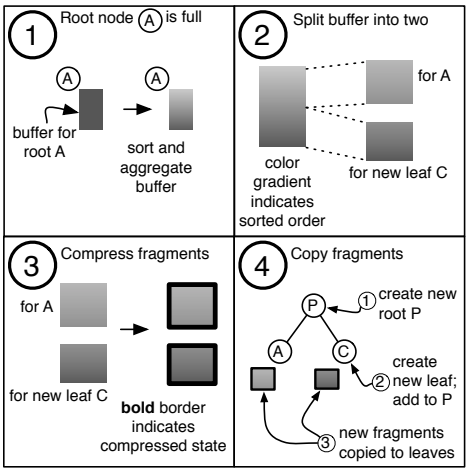

Figure 2: Split process for root node A: The root buffer is sorted and aggregated; the buffer is then split into two and each part compressed. One split is copied into a new node C. The new root P is created as the parent of A and C.

Algorithm 1 Emptying a node

```
# @param N: Node which must be emptied
def empty(N):
   if N is the root:
      if N is a leaf:
         # When the tree has only one node,
         # it is both the root and a leaf
         splitRoot() # see Figure 2
      else:
         spillRoot() # see Figure 3
   else:
      if N is a leaf:
         splitLeaf() # see Figure 4
      else:
         spillNode() # see Figure 5
```

Emptying the Root Node When the root becomes full, the PAOs in the root buffer are sorted by hash; a fast radix sort is used because 32-bit hashes are being sorted. After sorting, because PAOs with the same key appear consecutively in the sorted buffer, an aggregation pass is performed. This aggregates PAOs in the buffer to exactly one PAO per key. The sorted, aggregated buffer is then handled in two possible ways: if the root is also a leaf (if the tree only contains one node), then it undergoes the *split* procedure (Figure 2), otherwise, it undergoes the *spill* procedure (Figure 3).

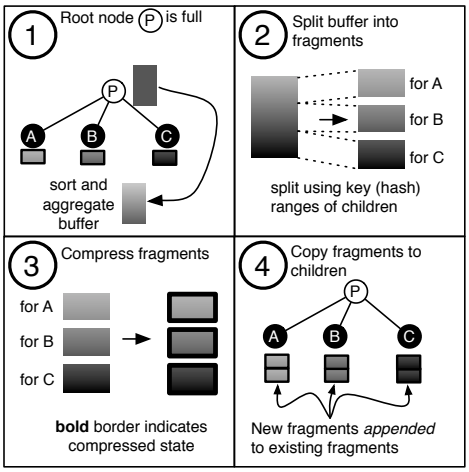

Figure 3: Spill process for root node A: The root buffer is sorted and aggregated; the buffer is then split into fragments according to hash ranges of children, and each fragment is compressed and copied into the respective children node.

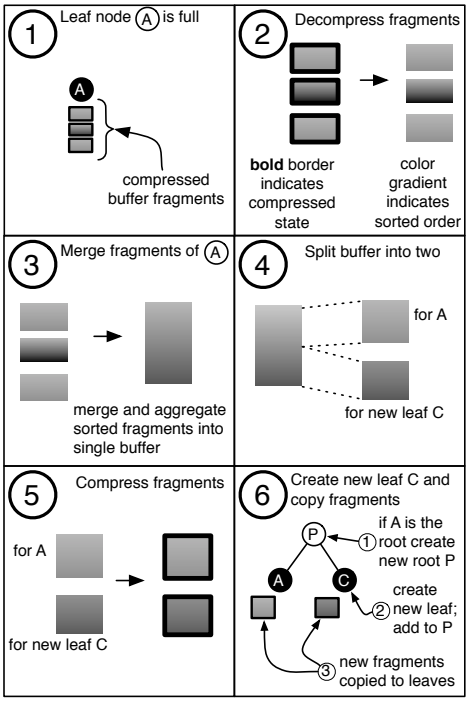

Figure 4: Split process for leaf node A: The compressed fragments in A are decompressed, merged (each is sorted), and divided into two splits. Both splits are compressed and one split copied into a new leaf node C.

Emptying Internal and Leaf Nodes Each root spill copies compressed buffer fragments into its children. Any node can spill until one or more of its children becomes full. A full child must first be emptied, before the parent can spill again. Internal and leaf nodes consist of compressed buffer fragments. During emptying, these

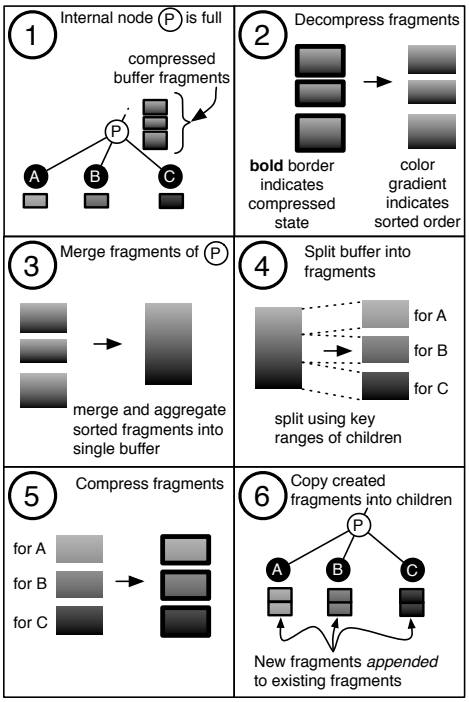

Figure 5: Spill process for internal node P: The compressed fragments in A are decompressed, merged (each is sorted), and split into fragments according to hash ranges of children. Each compressed fragment is copied into the respective child.

compressed fragments are first decompressed; since each decompressed fragment is already sorted, the fragments are then *merged* into a single buffer. During merging, PAOs with the same key are aggregated into a single PAO per key. If the node is a leaf, it is *split* (Figure 4), else *spilled* (Figure 5).

Handling hash collisions Ordering PAOs by hash values instead of the keys improves performance and reduces CPU use, but introduces the possibility of hash collisions. To aggregate a list of PAOs sorted by hash, in the event of no collisions, a single accumulator can be maintained and PAOs aggregated into it until the hash value changes. After this, the accumulator is written out to a separate aggregated list, and the process is repeated for the next set of PAOs.

When collisions occur, PAOs for colliding keys can occur in interleaved fashion. For example, the following sequence (containing hashes, sizes and PAOs) could occur: $\{h_a, s_1, (a,1)\}$, $\{h_b, s_2, (b,3)\}$, $\{h_a, s_3, (a,2)\}$, $\{h_c, s_4, (c,1)\}$, where a and b collide, i.e., $h_a = h_b \neq h_c$. Collisions are handled by maintaining accumulators for all colliding keys, and aggregating PAOs into the appropriate accumulator until the hash value changes. For the above sequence, an accumulator for a is first created, but since $h(a) = h(b)$,

but $a \neq b$, a second accumulator is created for b. These accumulators are stored in separate hash table. The second PAO for key a is aggregated into key a's accumulator. When the PAO for key c is read, $h(c) \neq h(a)$, so both accumulators for a and b are written out to the aggregated list, and the process continues.

Implementing `finalize()` Due to buffering, CBTs can have multiple PAOs for the same key buffered at different levels in the tree. When the final aggregated value is required, the CBT is finalized by performing a breadth-first traversal of the tree (starting from the root) and emptying each node using the procedure outlined in Algorithm 1. This causes all PAOs to be pushed down to the leaf nodes in the tree. At the end of this process, any possible duplicate PAOs would have been merged.

4 Implementation

We implement the CBT as a stand-alone aggregator in C++, allowing it to be integrated into different parallel-processing runtimes. The CBT can either be used as a library or as a service (uses ZeroMQ [4] for cross-language RPC). In either form, it can implement per-node aggregation in a distributed runtimes for MapReduce or stream-processing systems such as Muppet [32] or Storm.

4.1 Performance Optimizations

Asynchronous operations Core CBT operations such as compression, sorting/merging and emptying execute asynchronously. Queues are maintained for each of these operations and nodes are moved between queues after processing. For example, when a non-root node is full, it is inserted into the decompression queue. After decompression, it is queued for merging, where its fragments are merged and aggregated. Finally, the node is queued for emptying. All movement of nodes between queues is by reference only, with no copying of buffers involved. Queues also track dependencies to avoid cases such as a parent node spilling into an emptying child node.

Scaling the number of workers Structuring CBT operations as a pipeline of queues allows minimal synchronization during parallel execution (only required when adding/removing nodes from queues). Multi-core scaling is achieved by simply adding more worker threads per queue. We use a thread-pool for worker threads as work is bursty, especially for workloads with uniformly distributed keys where entire levels of nodes in the tree become full and empty within a short interval of time.

Multiple Root Buffers PAOs are inserted into the CBT by an insertion thread. The insertion thread blocks when the root buffer becomes full and remains blocked until the buffer is emptied into the nodes in the next level. Overall performance depends significantly on minimizing the blocking time for the insertion thread. Using multiple buffers for the root allows insertion to continue while buffers are sorted and aggregated and wait to be emptied. The inserter blocks only when all buffers are full.

Scaling the number of trees Due to the bursty nature of the work, a single tree, though multi-threaded, cannot utilize all CPUs on our node. Therefore, we use multiple CBTs, using hashes of the keys to partition PAOs between trees, such that any given key can occur only in one tree. We find it sufficient to restrict all threads belonging to one instance of the CBT to run on the same processor socket.

4.2 Memory Optimizations

Efficiency through Serialization CBTs improve memory efficiency through compression. As discussed, being able to compress buffers requires that PAOs are serialized into the buffers. Serialization, in fact, also improves memory-efficiency by avoiding many sources of memory overhead that "live" objects require. For example, if an uncompressed in-memory hashtable is used to store the aggregators, there are many sources of overhead:

1. Pointers: Because an intermediate key-value pair is hashed by key, the hashtable implementation stores a pointer to the key, at a cost of 8B (64 bit).
2. Memory allocator: Both keys and values are allocated on the heap, incurring overhead from the user-space memory allocator because: (a) sizes are rounded up to multiples of 8B to prevent external fragmentation (e.g. jemalloc [16] has size classes $8, 16, 32, 48, ..., 128$), (b) each allocation requires metadata; an allocator that handles small objects efficiently (e.g. jemalloc) reduces this overhead.
3. Hashtable implementation: unoccupied hashtable slots waste space to limit the load factor of the hashtable for performance. A memory-efficient implementation such as Sparsehash [21] minimizes this overhead and has a per-entry overhead of just 2.67 bits (at the cost of slightly slower inserts).

The CBT always stores PAOs in efficient[1] serialized form and avoids "live"-object overheads. "Live" PAOs also use C-style `structs` instead of C++ objects to

[1] e.g. Google Protocol Buffers (protobufs) serializes integers as *varints* which take one or more bytes allowing smaller values to use fewer bytes instead of the usual 4B

avoid virtual table overheads. Compared to hashtables, allocator overhead is minimal because memory is allocated for large buffers, both compressed and uncompressed (typical buffer sizes are hundreds of MB).

Column-Specialized Compression The CBT borrows the idea of organizing data by columns from column-store databases. This enables the use of specialized compression techniques to save memory. Recall that each buffer fragment in the CBT consists of tuples of the form ⟨`hash, size, serialized PAO`⟩; storing tuples column-wise results in three columns, each of which is compressed separately. For example, in each buffer fragment, the hashes are already sorted; therefore, we use Delta-encoding to compress the column of hashes. Because many of the PAOs are of similar size, we use Run-Length Encoding to compress the column of PAO sizes. For the column of actual PAOs, we use Snappy [20], a general-purpose compressor, but allow the user to override this with a custom compression algorithm.

5 Evaluation

In this section, we evaluate the CBT as an aggregator in streaming and batch modes and compare performance with uncompressed hashtable-based aggregators. Since the CBT trades off additional CPU use for memory savings, we introduce a cost model based on Amazon EC2 resource costs to study whether the saved memory is worth the cost of increased CPU consumption. Next, we perform a factor analysis to understand how CBT parameters and workload characteristics affect performance.

Setup The experiments use a server with two Intel X5650 processors (12 2.66GHz cores or 24 threads with SMT) with 48GB of DDR3 RAM. Each processor has 32KB L1 caches, 256KB L2 caches and a 12MB L3 cache. The system runs Linux 2.6.32 with the gcc-4.4 compiler (with -O3 optimization). Each experiment is repeated 12 times and the mean of the last 10 runs is reported with error bars denoting the standard deviation. We use the `jemalloc` [16] memory allocator; `jemalloc` and `tcmalloc` [19] both have good multi-threaded performance, but the latter shows inflated memory usage as it does not return freed memory to the kernel. Unless specified, we use the CBT with compression enabled using the Snappy [20] compressor; a buffer size of 30MB and a fan-out of 64 are used.

5.1 Applications

Recall that user-defined aggregate functions are specified using PAOs (Section 3.1). Next, the applications used in the evaluation are described.

Wordcount In this application, we count the number of occurrences of each unique word. We use this application with a series of synthetic datasets to understand the characteristics of the different runtimes and aggregators. The synthetic datasets are characterized by the number of unique keys, a measure of how often the key appears—aggregatability (for an application, we define the *aggregatability* of the dataset as the ratio of the size of the dataset to the aggregated size of the dataset), the average length of the keys and the distribution of key lengths. These datasets use randomly-generated keys, which is the worst-case for compressibility.

N-gram Statistics An n-gram is a continuous sequence of n items from a sequence of text. N-gram counting is useful in applications such as machine translation [38], spelling correction [28] and text classification [10]. This application computes n-gram statistics on 30k ebooks downloaded from Project Gutenberg [1] for different values of n. Each PAO contains a key-value pair: the n-gram and a 32-bit count. Merging PAOs simply adds the count values. The popularity of words in English follows a Zipf distribution, and this application tests the ability of aggregators to handle workloads where the aggregatability and key length is non-uniform across keys.

k-mer counting Counting k-mers, which are substrings of length k in DNA sequence data, is an essential component of many methods in bioinformatics, including genome assembly and for error correction of sequence reads [39]. In this application we count k-mers (for k=25) from Chromosomes 1 & 2 from human genomic DNA available at `ftp.ncbi.nlm.nih.gov/genomes/H_sapiens`. Each PAO consists of the k-mer as the key and a 32-bit count. Merging PAOs simply adds the count values, similar to n-gram. This application tests the ability of aggregators to handle datasets with a large number of unique keys. We use k-mer counting as a representative batch aggregation application.

Nearest Neighbor In this application, we detect similar images using the perceptual hashes [40] of 25 million images from the MIT Tiny Image dataset [47]. Perceptual hashes (PHs) of two images are close (as defined by a similarity metric like Hamming distance) if the images are perceptually similar according to the human visual response. Perceptual hashes are often used in duplicate detection.

This application consists of two MapReduce jobs: the first job clusters images that have the same PH prefix and the second job performs a brute-force nearest-neighbor

search among these clusters. For the first job, a PAO consists of a PH-prefix as key and a list of `(image ID, perceptual hash)` tuples, which denote the images in that cluster, as value. Merging two PAOs with the same key combines their image lists. From an input image and its hash (e.g. A, 563), a PAO is created whose key is a prefix of the PH (e.g. 56) and whose value is the image's file name. Therefore, PAOs for images with the same prefix (e.g. 561, 567), which by definition are perceptually similar, can be merged by merging the file lists from the PAOs. This job has an expensive reduce function and large PAOs.[2] The second job generates all possible pairs of images from each cluster; the key is the image ID and the value is a `(neighbor ID, distance)` tuple. These are merged by choosing neighbors with the least distance.

5.2 Comparison with Hashtable-based Aggregators

Workload generator To model a stream-based system, we use a multi-threaded workload generator that generates streams of application-specific PAOs that are inserted into the aggregator for grouping and aggregation. To generate the workload, the workload generator can either use an input dataset file or generate randomized data according to input parameters. We run the workload generator on the same machine as the aggregator and link the aggregator libraries into the workload generator. For all experiments, we find that using four generator threads is sufficient to saturate all aggregators.

The workload generator also periodically *finalizes* the aggregators. For hashtable-based aggregators, this has no effect since they always maintain up-to-date aggregated values for all keys. CBTs, however, can have multiple PAOs per key buffered, so finalization causes these to be merged to a single PAO per key.

Hashtable-based Aggregators For comparison with the CBT, we also implemented two stand-alone hashtable-based aggregators: 1) *SparseHash (SH)*, which permits only serial access but uses Google's `sparse_hash_map`, an extremely memory-efficient hashtable implementation, and 2) *ConcurrentHashTable (CHT)*, which allows concurrent access using the `concurrent_hash_map` from Intel's Threading Building Blocks [26].

Unlike the CBT, which maintains intermediate key-value pairs in compact, serialized form before being compressed, SH and CHT maintain each (unique) intermediate key-value pair unserialized. We consider two representations for the intermediate key-value pairs: (a) statically sized, where the intermediate key-value pair is stored in-place in a statically-sized C++ `struct`, and (b) dynamically sized, which uses pointers. The former approach (called SH and CHT respectively) requires the `struct` to be sized to the largest key-value pair, but avoids heap allocation overheads, whereas the latter approach (called SH-ptr and CHT-ptr respectively) allocates the exact amount of memory required for each intermediate key-value pair. For the CBT, we also show performance with compression disabled (BT).

[2]This basic method does not find images whose hashes are close by Hamming distance but differ in higher-order bits (e.g. 463 and 563). Therefore, we repeat the process after rotating the PH values for each image (635 and 634 share the same prefix). This works because the Hamming distance is invariant under rotations.

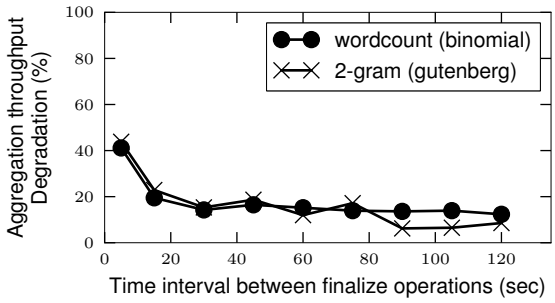

Figure 7: Effect of time interval between finalize operations on aggregation throughput

A single instance of each aggregator uses a different amount of CPU. For fair comparison, we use the same amount of CPU overall by using multiple instances for the CBT and SparseHash and partitioning the PAO stream between the instances (e.g. by using a hash function on the PAO key). Therefore, we use 5 instances of the CBT and 20 instances of SparseHash.

First, we compare the aggregators with finalization performed only once at the end of the input dataset. This is equivalent to batch-mode execution. Figure 6 shows these results, and we make the following observations: (a) For all applications, CBT consumes the least amount of memory; (b) the performance of BT is better than the alternatives (except for the application wordcount(uniform) which is a special case with words of exactly the same length); (c) BT always offers better throughput than CBT; (d) CBT consumes significantly less memory than BT if the dataset is compressible: the randomized text and ebook datasets are not highly compressible, but the k-mer and nearest-neighbor are. Many real-world datasets, such as URLs, DNA sequences etc., are compressible; finally, (e) for the hashtable-based aggregators, no intermediate key-value pair representation (static or dynamic) consistently outperforms the other.

As the frequency of finalization increases, hashtables remain unaffected, but CBT performance decreases. Fig-

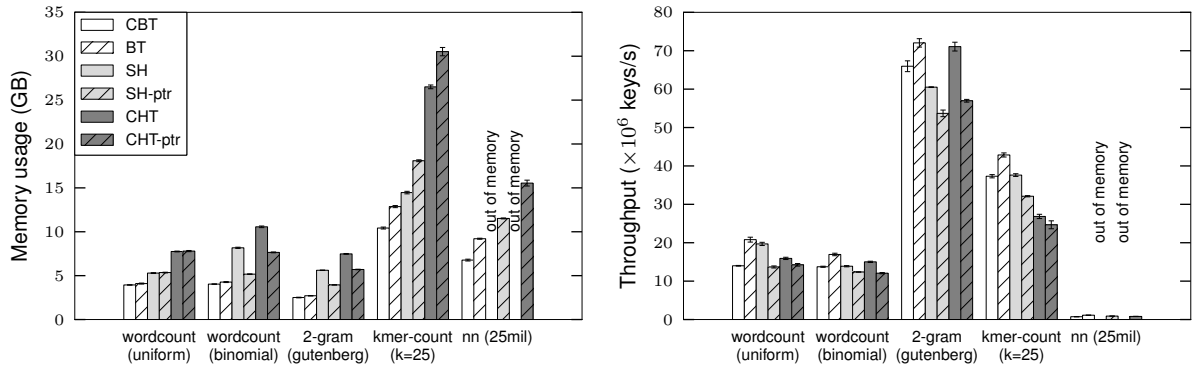

Figure 6: Comparison of various aggregators

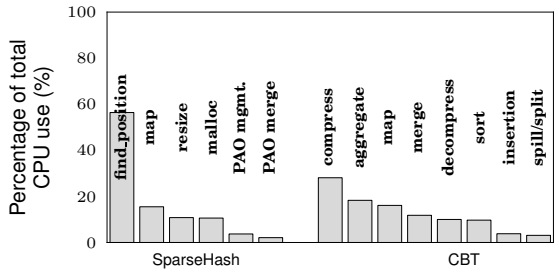

Figure 8: Comparison of CPU use by SparseHash (20 instances) and CBT (4 instances): For SparseHash, `find_position`: find an unassigned index in a closed hashtable, `map`: map-side processing (tokenizing of keys etc.), `resize`: hashtable resizing when full, `malloc`: memory allocation, `PAO mgmt.`: creation of PAOs for insertion into hashtable, `PAO merge`: merging of PAOs with the same key. For CBT, each bar refers to the respective operation performed on buffer fragments in CBT nodes.

ure 7 shows the effect of finalization frequency on degradation of aggregation performance for two applications: WordCount (Binomial) and 2-gram (Gutenberg). It can be seen that while frequent finalization (once every 5s) can lead to high degradation, more realistic inter-finalize intervals lead to between 10-20% of degradation in aggregation throughput.

Finally, intuitively, it seems that hashtable-based aggregators should always outperform CBTs, given that aggregation using CBTs entails compression, sorting, etc. However, especially for memory-efficient hashtables, the comparison is hard to intuit because aggregation using hashtables involves overheads not present in the CBT. Figure 8 shows the breakup of CPU use for operations involved in aggregation using both hashtables and CBTs.

For clarity, we briefly explain how SparseHash works: The hashtable is implemented as an array of fixed-sized "groups". Each hashtable operation (e.g. insert, find) is passed on to the group responsible for the index returned by the hash of the key. Each group consists of a bitmap, that stores whether a particular index in the group is assigned or not, and a vector that stores values for assigned indices **only**. This allows the hashtable to use less memory (2.67 bits) for each entry, wasting little memory for unassigned indices. The savings in memory result in slower inserts, as indicated by the high CPU use of `find_position`. SparseHash uses quadratic probing for collision resolution; during an insert, multiple string comparisons might be performed along the probe sequence. Like other hashtables, SparseHash also requires resizing when the load factor (for a closed hashtable, the proportion of occupied indices) becomes too high. Sparsehash also incurs higher allocator overhead than regular hashtables, as unassigned indexes are not pre-allocated and have to be allocated on inserts.

5.2.1 Cost model

Estimating the cost-benefit of using the CBT requires a cost model for the resources it consumes. The model must account for the total cost of operation including purchase and energy costs. Instead of synthesizing a model from component costs, inspired by an idea in a blog post [36], we analyze the publicly available pricing of different Amazon EC2 instances, which have varying amounts of CPU, memory and disk resources, to estimate the unit cost of each kind of resource as priced by Amazon as alluded to in Section 1. We describe the derivation next.

Let A be an $m \times 3$ matrix with each row containing the number of units of CPU, memory and storage available in a type of instance. There is one row for each of the m different types of EC2 instances (e.g. Standard Small, Hi-memory Large etc.). Let b be an $m \times 1$ matrix representing the hourly rates of each kind of instance, and let $x = [c, m, d]^\top$ be the per-unit-resource rates for CPU, memory and disk respectively. Solving

Application	(x,y)	Total cost (¢)						Savings
		CBT	BT	SH	SH-ptr	CHT	CHT-ptr	
wordcount (uniform)	100, 10	0.43	**0.28**	0.31	0.43	0.50	0.57	9.6%
wordcount (binomial)	100, 10	0.41	**0.31**	0.51	0.46	0.64	0.65	32.6%
2-gram (gutenberg)	105, 80.5	0.53	**0.43**	0.70	0.67	0.86	0.92	35.8%
k-mer (genomic)	394, 6.1	**0.53**	0.56	0.68	0.94	1.50	1.84	22%
nn (25mil)	116, 5.3	**0.15**	0.18	-	0.25	-	0.33	40%

Table 4: Dataset Parameters and Costs using Amazon EC2 cost model: x represents the number of unique keys in the dataset (in millions) and y represents the average number of occurrences. The cost benefit obtained by using the better of CBT/BT (bold) over the best option among alternatives (underlined).

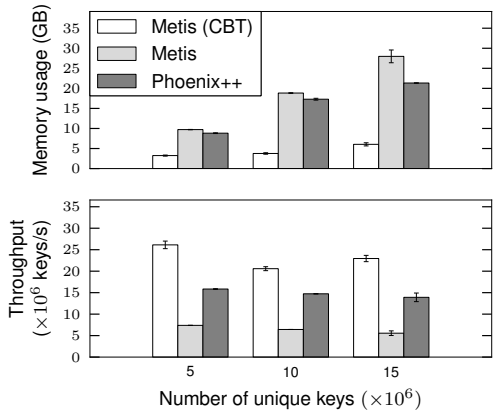

Figure 9: Comparison of CBT with aggregators in Metis and Phoenix++: All systems sort final results by count and display the top 5 words. The synthetic datasets consist of an increasing number of unique keys, with average key length 16B, and each key repeating 1000 times.

for x in $A.x = b$ using a least squares solution that minimizes $\|b - A.x\|^2$ yields the per-unit-resource costs. Using prices from April 2012, this yields hourly costs of 1.51¢ per ECU (Elastic Compute Unit), 1.93¢ per GB of RAM and 0.018¢ per GB of storage. Table 4 shows the corresponding costs predicted by our model along with the benefits of using the CBT over its alternatives.

Batch-mode comparison with MapReduce Aggregators The CBT can also be used as a batch-mode aggregator. For comparison, we replace the default aggregator in the Metis [37] MapReduce framework, which uses a hash table with a B+ Tree in each entry to store intermediate data, with the CBT. We use a set of synthetic datasets with an increasing number of unique keys to compare the growth in memory of the independent aggregator pipeline with the partitioned aggregator pipeline.

The number of mappers (and aggregators) for (default) Metis and Phoenix++ is set to 24, which is equal to the number of logical cores on our system. Metis and Phoenix++ performance scales linearly up to 24 logical cores, and both systems are able to utilize all available cores. For Metis-CBT, we use only 5 instances of the CBT, with each instance configured with 16 worker threads, as this is able to saturate all cores on the system. Figure 9 shows that Metis with CBT outperforms Metis and Phoenix++, while requiring up to 5× and 4× less memory. Recall that simply compressing the hashtables used in Metis or Phoenix++ will not yield the same benefits, since each hashtable entry is typically too small to yield significant memory savings.

5.3 CBT Factor Analysis

In this section, we use microbenchmarks to evaluate the performance of the CBT for varying workload characteristics and the effect of different CBT parameters. We first evaluate the impact of CBT design features and configuration parameters on aggregation performance and memory consumption. Finally, we evaluate the effects of workload characteristics.

5.3.1 Design features

We analyze the performance of the CBT by considering its features in isolation. For each feature, we show its incremental impact on aggregation throughput and memory consumption in Figure 10. Gains/losses are relative to previous version and not to the baseline. For reference, the memory use and throughput of the SparseHash-dynamic (SH-ptr) aggregator is also shown. A synthetic dataset with 100 million unique keys, each repeating 10 times, with binomially ($p = 0.15$) distributed key-lengths between 6 and 60B, is used.

The baseline CBT consists of a single instance of the CBT. It has 4 worker threads: one each for compression, sorting, aggregation and emptying. The basic CBT consumes about 50% less memory compared to SH-ptr. It avoids overheads associated with allocating small objects on the heap (by always allocating large buffers) and avoids storing pointers for each PAO (by serializing the

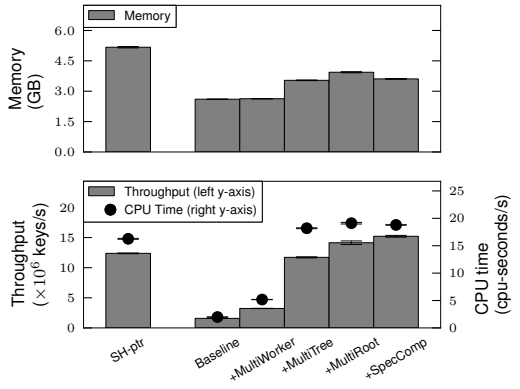

Figure 10: Effects of CBT Optimizations

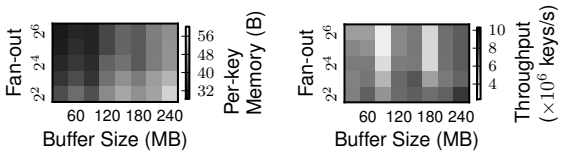

Figure 11: Effects of tree fan-out and buffer size

PAO into the buffer) in addition to compression. Aggregation throughput of the baseline CBT is about 87% less than that of SH-ptr.

By increasing the number of worker threads to 16, CBT parallelism increases (overall CPU use increases 2.6×) and improves throughput by 1.5×. Due to the burstiness of the work generated by the CBT, adding more CBT instances allows full use of the multi-core platform. CPU use increases 3.5×, boosting aggregation throughput by 3.6×. Per-instance static overhead increases the memory use by 1.3×. Using multiple root buffers ($n = 4$ in this case) allows insertion to continue even when the root is being emptied. "MultiRoot" increases throughput by nearly 21% at the cost of 11% additional memory (increasing the number of buffers beyond 4 did not increase performance enough to justify the increase in memory use). Finally, specialized compression for each column, "SpecComp", which uses delta encoding for the hashes-column, run-length encoding for the sizes-column and Snappy for the PAOs, improves the effectiveness of compression, reducing memory use by a further 8%.

Next we consider how the performance and memory consumption of the CBT depend on system parameters such as the node buffer size and the tree fan-out.

5.3.2 CBT Parameters

Variation with buffer size and fan-out of CBT Although the CBT avoids many overheads associated with hashtable-based aggregation, it incurs memory overhead

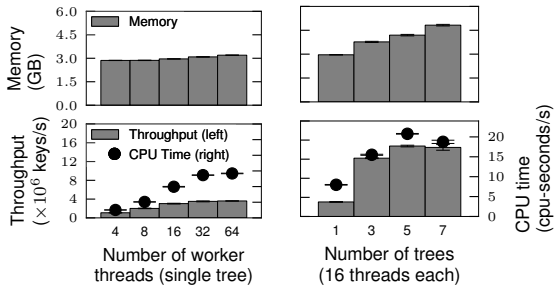

Figure 12: Scaling aggregation throughput

when serialized PAOs with the same key occur at multiple levels of the tree as a result of lazy aggregation.

This overhead depends on certain CBT parameters: we provide intuition for the dependence of both memory consumption and aggregation performance on these parameters. Figure 11 shows heat-maps for memory use and aggregation throughput for variations in buffer size and fan-out of the tree. Darker colors imply lower memory overheads and higher aggregation performance respectively and vice versa. In general, increasing buffer size increases aggregation performance, but also increases memory overhead. The reason for this is that larger buffers allow less frequent spilling (improving performance), but provide a greater opportunity for keys to repeat (increasing memory overhead).

The trend with fan-out is similar: increasing fan-outs decrease (colors lighten) memory overhead as well as aggregation throughput. This is because an increase in fan-out results in a decrease in the height of the tree. Shallower trees provide less opportunity for keys to repeat at different levels of the tree (decreasing memory overhead), but also lead to more frequent spilling (decreasing performance).

Therefore, adjusting the buffer size and fan out when configuring the CBT allows the user to trade off throughput for memory efficiency, or vice versa, as desired.

Scalability Figure 12 shows how the memory use and throughput of the CBT scales with an increasing number of worker threads (left), and by partitioning keys across an increasing number of trees (right). In the rightmost graph, each tree uses 16 worker threads. The striking throughput improvement from using multiple trees arises because they help smooth out the otherwise extremely-bursty workload imposed upon the CBT; without additional trees, the CBT alternates between periods of idle and periods of maximum CPU use. With the additional trees, the cores are occupied nearly full-time. Partitioning keys across more trees adds additional root buffers, increasing memory use modestly.

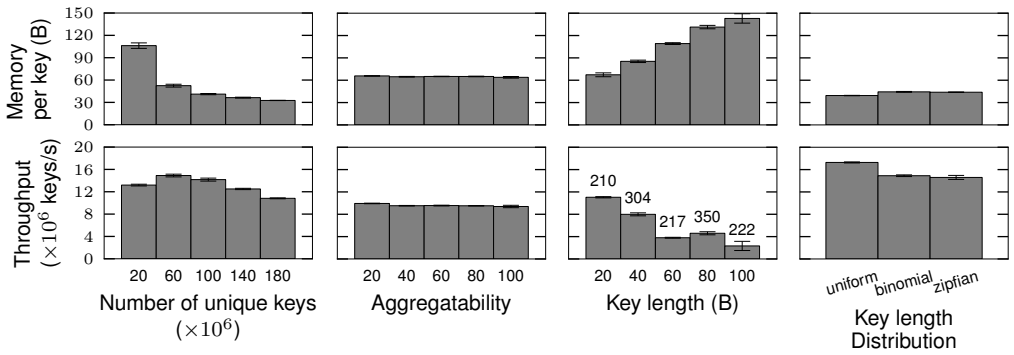

Figure 13: Evaluating the CBT with Microbenchmarks

5.3.3 Workload Properties

Here we evaluate how workload properties affect the performance of the CBT. We consider the number of unique keys, the aggregatability (for wordcount in the following experiments), the size of the PAO, and distribution of key length. Figure 13 shows the per-key memory consumption and aggregation throughput.

Aggregated data size We use synthetic datasets with an increasing number of unique keys while holding other properties constant. Figure 13 shows that the static memory use of the CBT causes per-key use to be high for (relatively) small number of keys, but this cost gets amortized with increasing number of unique keys.

Aggregatability We examine memory efficiency as a function of increasing aggregatability by using a progressively larger number of total keys with a fixed number of unique keys. With hashtable-based aggregation, increased aggregatability does not require more memory since a single PAO is maintained per key (and wordcount's PAOs do not grow with aggregation). Figure 13 shows that even for a CBT, despite buffering, per-key memory use does not increase with aggregatability.

PAO size Datasets with increasing key size are used, with the rest of the parameters remaining constant. The per-key memory use increases with increasing key-size, as expected. While the throughput appears to drop with increasing key-size in Figure 13, the throughput in terms of MB/s shown above the bars, does not share this trend.

Key length distribution We use datasets with 100 million unique keys, each occurring 10 times, with varying distributions on key length: uniform, binomial and Zipf; the average length of each word is around 10B. The throughput for binomial and Zipf are similar, with uniform being marginally higher owing to better cache use (each cache line fits 4 serialized PAOs) which improves performance during sorting and aggregation.

6 Related Work

GroupBy-Aggregate Yu et al.'s work [48] discusses the generality of the GroupBy-Aggregate operation across a range of programming models including MapReduce and Dryad [27]. Graefe [22] and Chaudhuri [13] survey sort and hash-based techniques for aggregation in the batched context for optimization of database queries. There is also a significant amount of previous work in online aggregation in databases [24] that returns early results and on-the-fly modification of queries. The CUBE SQL operator was proposed by Gray et al. to support online aggregation by pre-computing query results for queries that involved grouping operations [23]. Iceberg-CUBE extended the CUBE operation to include user-specified condition to filter out groups [9]. Kotidis et al. propose R-trees to support Cube operations with storage reduction and faster updates [30].

Stream-processing systems such as Muppet [32], which provides a MapReduce-like interface to operate on streams, and Storm [3], also support interactive queries by pre-computing aggregates. These systems allow fully user-defined aggregate functions to be set up as aggregators that are updated in near real-time with input data streams. Other stream-processing systems, including SPADE [18], provide similar options for aggregation. The CBT can be used as a drop-in replacement for aggregators in these systems.

Other work that has considered the optimization of the aggregator include Tenzing [12], which is a SQL query execution engine based on MapReduce and uses hashtables to support SQL aggregation operations, and One-pass MapReduce [33]. Phoenix++ and its predecessors [46, 42], Metis [37], Tiled-MapReduce [14] and MATE [29] are shared-memory MapReduce implementations that also focus on the design of aggregation data structures. The CBT is able to match or better the perfor-

mance of these aggregators, while achieving significantly lesser memory consumption.

Write-Optimized Data Structures In this paper, a crucial insight is that prior work in write-optimized data structures is applicable to aggregation using compressed memory. Log-structured approaches are used because in-place update data structures such as the B-Tree and hashtables [17] do not provide the required write performance. Sears et al. differentiate between ordered and unordered log-structured approaches [45]. Among ordered approaches, LSM Trees [41] and variants [45] are used in many real-world key-value stores [11, 31]. Buffer trees [7] offer better write performance, but only good amortized read performance (I/Os per read is low, but latency can be high due to buffering). The CBT adopts the buffer tree because lazy aggregation requires fast writes and ordering, but not low-latency reads.

Compression in Databases There is a significant body of work involving compression in databases. Chen et al. compress the contents of databases, and derive optimal plans for queries involving compressed attributes [15], and Li et al. consider aggregation algorithms in a compressed Multi-dimensional OLAP databases [34]. We believe that the CBT is a promising candidate for implementing aggregation within RDBMS systems.

7 Future Work

External aggregation Although the CBT decreases memory use in aggregated data compared to existing approaches, datasets may still be too large to fit in memory. One possible extension of this work is to use ideas from the original buffer tree data structure that inspired the CBT: handling external aggregation, using fast storage such as SSDs. Possible heuristics include (a) maintaining lower levels of the tree, closer to the leaf, on disk, while higher levels are in remain compressed memory, or (b) compressing sub-trees with high aggregation activity in memory while keeping the remaining sub-trees on disk.

Supporting random access The CBT does not currently support random access to aggregated PAOs by key. To support this, the buffers containing PAOs for the queried key at different levels in the tree have to be identified and decompressed. Then, the PAOs of the queried key have to be located within the decompressed buffers and aggregated. This overhead makes it unlikely that CBTs can support random access faster than hashtables, which require just a few memory accesses.

8 Conclusion

This paper introduced the design and implementation of a new structure for memory-efficient, high-throughput aggregation: the Compressed Buffer Tree (CBT). The CBT is a stand-alone aggregator that can be used in different data-parallel frameworks. Our evaluation results show that when used as an aggregator for streaming data, the CBT uses up to 42% less memory than an aggregator based on Google SparseHash, an extremely memory-efficient hashtable implementation, while achieving equal or better throughput. CBTs can also be used in MapReduce runtimes. Substituting the default aggregator in the Metis MapReduce framework with the CBT, enables it to operate using 4-5$\times$ less memory and run 1.5-4$\times$ faster than default Metis and Phoenix++, another modern in-memory MapReduce implementation.

The primary goal of the CBT is to allow efficient aggregation on compressed data. Hashtables offer the *read-modify-update* paradigm for aggregation where up-to-date aggregates are always maintained for all keys. This paradigm makes it nearly impossible to keep data compressed since it may be accessed at any time. The CBT is built on the realization that aggregation need not be eager and aggregates can be lazily computed. With this relaxation, the CBT maintains aggregate data in compressed form (increasing memory efficiency), and limits the number of decompression and compression operations (increasing aggregation throughput).

Acknowledgments

This work was funded in part by Intel via the Intel Science and Technology Center on Cloud Computing (ISTC-CC). We would like to thank the SoCC reviewers, and Vishal Gupta for their feedback, and Frank McSherry for shepherding this paper.

References

[1] Project Gutenberg. www.gutenberg.org.

[2] LZO. oberhumer.com/opensource/lzo.

[3] Storm. storm-project.net.

[4] ØMQ (zeroMQ). zeromq.org.

[5] B. Abali, H. Franke, X. Shen, D. E. Poff, and T. B. Smith. Performance of Hardware Compressed Main Memory. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 73–81. IEEE Computer Society, 2001.

[6] A. Aggarwal and S. Vitter, Jeffrey. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[7] L. Arge. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica*, 37(1): 1–24, 2003.

[8] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-Oblivious Streaming B-Trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 81–92. ACM, 2007.

[9] K. Beyer and R. Ramakrishnan. Bottom-up Computation of Sparse and Iceberg CUBE. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 359–370. ACM, 1999.

[10] W. B. Cavnar and J. M. Trenkle. N-Gram-Based Text Categorization. In *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[12] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing: A SQL Implementation on the Mapreduce Framework. In *Proceedings of the VLDB Endowment*, volume 4, pages 1318–1327, 2011.

[13] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 34–43. ACM, 1998.

[14] R. Chen, H. Chen, and B. Zang. Tiled-Mapreduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 523–534. ACM, 2010.

[15] Z. Chen, J. Gehrke, and F. Korn. Query Optimization in Compressed Database Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 271–282. ACM, 2001.

[16] J. Evans. A Scalable Concurrent `malloc(3)` Implementation for FreeBSD.

[17] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible Hashing - a Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.*, 4(3):315–344, Sept. 1979.

[18] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the System S Declarative Stream Processing Engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134. ACM, 2008.

[19] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. `goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[20] Google. Snappy. `code.google.com/p/snappy`, .

[21] Google. Sparsehash. `code.google.com/p/sparsehash`, .

[22] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[23] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, Jan. 1997.

[24] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182. ACM, 1997.

[25] S. Huddleston and K. Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17(2): 157–184, 1982.

[26] Intel Corporation. Intel Threading Building Blocks. `www.threadingbuildingblocks.org`.

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72. ACM, 2007.

[28] A. Islam and D. Inkpen. Real-Word Spelling Correction using Google Web IT 3-grams. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, volume 3 of *EMNLP '09*, pages 1241–1249. Association for Computational Linguistics, 2009.

[29] W. Jiang, V. T. Ravi, and G. Agrawal. A Map-Reduce System with an Alternate API for Multi-core Environments. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 84–93. IEEE Computer Society, 2010.

[30] Y. Kotidis and N. Roussopoulos. An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 249–258. ACM, 1998.

[31] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[32] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-Style Processing of Fast Data. *Proc. VLDB Endow.*, 5(12):1814–1825, Aug. 2012.

[33] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A Platform for Scalable One-Pass Analytics using MapReduce. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, pages 985–996. ACM, 2011.

[34] J. Li, D. Rotem, and J. Srivastava. Aggregation Algorithms for Very Large Compressed Data Warehouses. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 651–662. Morgan Kaufmann Publishers Inc., 1999.

[35] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for

Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278. ACM, 2009.

[36] H. Liu. huanliu.wordpress.com/2011/01/24/the-true-cost-of-an-ecu/.

[37] Y. Mao, R. Morris, and F. Kaashoek. Optimizing Mapreduce for Multicore Architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, 2010.

[38] J. B. Mariòo, R. E. Banchs, J. M. Crego, A. de Gispert, P. Lambert, J. A. R. Fonollosa, and M. R. Costa-jussà. N-gram-based Machine Translation. *Comput. Linguist.*, 32(4):527–549, Dec. 2006.

[39] P. Melsted and J. Pritchard. Efficient Counting of k-mers in DNA Sequences using a Bloom Filter. *BMC Bioinformatics*, 12(1):1–7, 2011.

[40] V. Monga and B. L. Evans. Robust Perceptual Image Hashing Using Feature Points. In *PROC. IEEE Conference on Image Processing*, pages 677–680, 2004.

[41] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33 (4):351–385, June 1996.

[42] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24. IEEE Computer Society, 2007.

[43] L. Rizzo. A Very Fast Algorithm for RAM Compression. *SIGOPS Oper. Syst. Rev.*, 31(2):36–45, Apr. 1997.

[44] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[45] R. Sears and R. Ramakrishnan. bLSM: a General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228. ACM, 2012.

[46] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for Shared-Memory Systems. In *Proceedings of the Second International Workshop on MapReduce and its Applications*, MapReduce '11, pages 9–16. ACM, 2011.

[47] A. Torralba, R. Fergus, and W. Freeman. 80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(11):1958 –1970, Nov. 2008.

[48] Y. Yu, P. K. Gunda, and M. Isard. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*. ACM, 2009.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10. USENIX Association, 2010.

[50] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inf. Theor.*, 23(3): 337–343, Sept. 1977.

[51] J. Ziv and A. Lempel. Compression of Individual Sequences Via Variable-Rate Coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.