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 Pittsburgh

Abstract

Memory is rapidly becoming a precious resource in many data processing environments. This paper introduces a new data structure called a *Compressed Buffer Tree* (CBT). Using a combination of buffering, compression, and lazy aggregation, CBTs can improve the memory efficiency of the GroupBy-Aggregate abstraction which forms the basis of many data processing models like MapReduce and databases. We evaluate CBTs in the context of MapReduce aggregation, and show that CBTs can provide significant advantages over existing hash-based aggregation techniques: up to 2× less memory and 1.5× the throughput, at the cost of 2.5× CPU.

1 Introduction

This paper introduces a new data structure designed to enable the efficient application of compression to aggregation workloads, the *Compressed Buffer Tree* (CBT). We demonstrate its effectiveness by using it to implement the GroupBy-Aggregate abstraction found in many data processing models like MapReduce [16] and databases [25]. Given a dataset consisting of records, this abstraction partitions the records into groups according to some key, then executes the aggregation function on each group.

We choose this focus for two reasons: (1) the rising costs of memory in terms of money, power, and speed, and (2) the importance of the GroupBy-Aggregate abstraction in a range of distributed programming models. Improving the memory efficiency of this abstraction can drastically reduce the costs of execution. In this paper, we focus on the implementation of GroupBy-Aggregate in MapReduce, which has become a critical analytical tool for data-intensive applications, from machine learning [6], to text search [5], and more.

Memory is a valuable commodity in datacenters. DRAM is expensive and an expensive consumer of power [29]. Memory accesses are a common bottleneck for high-performance applications [49]. With the number of cores per socket growing faster than the memory capacity per socket, memory is increasingly scarce [37].

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 proportional resource costs (# resource units × per-hour unit resource cost); the per-hour unit resource costs are 1.51¢ (1 Elastic Compute Unit), 1.93¢ (1GB RAM) and 0.018¢ (1GB storage); analysis detailed in §5.3.

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, both in terms of the proportion of total instance cost as well as the per-unit-resource cost. In other words, with Amazon's standard configurations, it is far cheaper to double the CPU capacity than to double the memory capacity.

Given these facts, this paper explores the use of compression and other techniques to drastically improve the memory efficiency of the GroupBy-Aggregate operation in the context of MapReduce.

Our resulting techniques can be viewed as either reducing overall memory requirements (saving cost), or improving the amount of aggregation that can be performed (saving resources such as bandwidth and reducing load on the reducer nodes).

Recall that MapReduce operates in four stages: it applies a `map` function to incoming keys, groups identical keys together (usually by sorting), applies an intermediate aggregation function to these groups (combining), distributes each key to its final reducer node, and applies final cluster-wide aggregation (reduction) at the reducer nodes. Essentially, both the combiner and the reducer implement the GroupBy-Aggregate operation. Prior work has shown that the performance of a MapReduce job is largely affected by the characteristics of this operation [45].

For many applications, hash-based grouping improves

performance [11, 34, 53] because these applications require only unsorted grouping. Unfortunately, hash-based aggregation, more so than sorting, is a critical consumer of memory in MapReduce: it keeps many key-value pairs in memory during aggregation in order to efficiently merge records with the same key. The performance of hash-based aggregators also depends significantly on the amount of memory available. Insufficient memory can limit the amount of aggregation that can be performed [34] or lead to job failure [11].

To solve these challenges, we present a new, memory-efficient hash-based aggregator called the Compressed Buffer Tree (CBT). A CBT stores the partially aggregated contents in compressed form in memory to reduce memory consumption. Compressing individual key-value pairs has high overhead, so the CBT compresses larger buffers for efficiency. To avoid compressing and decompressing buffers too often, the CBT organizes the compressed buffers in memory as a B-tree using ideas from buffer trees [8]. We describe the CBT in Section 3.

The CBT is implemented in a new MapReduce library called Minni written in C++. By structuring aggregators in the form of pipelines, Minni enables fast, scalable execution on multi-core platforms and allows new aggregators to be constructed in a modular fashion. We describe Minni in further detail in Section 4.

2 Overview

This section motivates why GroupBy-Aggregate is a particularly important focus for memory reduction and why existing aggregators fundamentally have high overhead.

2.1 The "Memory Capacity Wall"

Several reports suggest that the relative cost of memory (vs compute) is likely to grow in the future. As one example, Lim et al. project that the gap between the number of cores and memory capacity per socket will continue growing, effectively decreasing the amount of memory available per core [37]. Although Compressed Buffer Trees are a generally applicable technique, as we describe below, the GroupBy-Aggregate operation is one of the major memory consumers in MapReduce-style processing, and so we specifically focus in this paper on improving its memory use.

2.2 Memory-Efficient Aggregation

MapReduce applications involve one or more GroupBy-Aggregate operations. A GroupBy-Aggregate operation can occur as a combiner for map-side pre-aggregation, and it always occurs on the reduce-side to perform final aggregation (see Figure 1). The operation takes key-value pairs, either from the local map operation (map-side) or from remote mapper processes (reduce-side). GroupBy-Aggregate first groups the pairs into partitions based upon

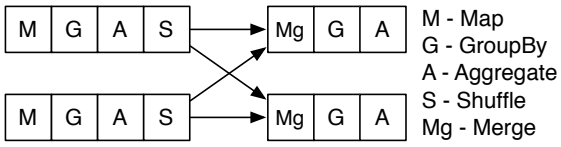


Figure 1: MapReduce flow diagram: GroupBy-Aggregate can involve two separate operations (as with sort followed by aggregate) or a single, combined operation (hash and accumulate).

the key, and then applies an aggregation function to each partition.

The design of an efficient GroupBy-Aggregate operation is the major focus of our paper because the GroupBy-Aggregate operation is fundamental to the overall performance of MapReduce applications. For brevity, we refer to the GroupBy-Aggregate operation as *aggregation* and the operator itself the *aggregator*.

In this paper, we focus on in-memory aggregators that do not use external storage as scratch space. In general, aggregation in memory is faster than aggregation using disk which often requires multiple passes. However, if the aggregated size of the data is too large for memory, then external aggregation involving multiple aggregation passes may be required, which we leave for future work.

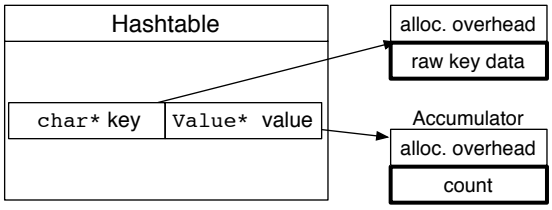
The primary design goals for our aggregator are:

1. **Memory-efficiency**: reduce the memory capacity overhead associated with aggregation in memory.
   Rationale: Avoid spilling state to (slow) disk and reduce DRAM costs.
2. **High aggregation throughput**: maximize the rate of aggregation of intermediate key-value pairs.
   Rationale: a significant portion of the execution time of a MapReduce job is spent in aggregation, hence the performance of the aggregator is critical.

2.3 Sort vs. Hash-based Aggregation

The Sort-merge Approach. Hadoop, a popular MapReduce implementation, stores intermediate key-value pairs emitted by the map operator in a memory buffer. When the buffer reaches a threshold size, the contents are partitioned (for the different reducers), and each partition is sorted, aggregated, and then spilled to disk. After the map task has processed all input, the spill files are merged to form a single sorted file for each partition.

For applications that require only map/group-by, and need not output keys in sorted order, recent work has shown that hash-based aggregation almost always outperforms sorting [53], in MapReduce, Dryad [31], and related aggregation contexts such as parallel databases [18]. Sort-merge has two problems: (1) it *orders* keys as a side-effect of grouping them which leads to unnecessary computation; and (2) the amount of memory required for



(a) **Memory overheads in hashtable-based aggregation**

Allocator	Per-entry memory (B)	
	`std::unordered_map`	`sparse_hash_map`
hoard [9]	64.9	67.8
tcmalloc [21]	57.2	43
jemalloc [20]	58.1	**41**

(b) **Per-entry memory consumption for different allocator and hashtable combinations (Unique keys inserted: 2^{26}, keys: 8B strings, values: 4B integers; g++-4.4 compiler on 64-bit system**

Figure 2: Hashtable-based aggregation

sorting is proportional to the total number of intermediate key-value pairs in the buffer rather than the number of distinct keys. This is inefficient for highly aggregatable workloads (where the dataset is much larger than the aggregated size) because it either reduces aggregation or requires multiple passes. This problem motivated the consideration of hash-based aggregators for MapReduce [11, 34].

The Hash-based Approach. Hash-based aggregation is common in RDBMSs, and recent work has begun to explore its application to MapReduce [11, 34].

Aggregation using hashing works as follows: a data structure such as a hashtable stores one "accumulator" for each key. Intermediate key-value pairs are then hashed by key and accumulated. Finally, the aggregated key-value pairs are read iteratively from the hashtable and transferred to the reducers.

Unfortunately, aggregation using a hashtable incurs high memory overhead per entry in the hash map. Before describing our alternative—Compressed Buffer Trees—we briefly walk through the overheads of hash-based aggregation and show how to shrink them in order to have a fair basis for comparison. Figure 2a shows the implementation of hash-based aggregation for wordcount using a hashtable, which maps string keys to accumulators, along with associated overheads. The overheads can be classified as follows:

1. Key-value pointers: pointers to the key and the accumulator are stored in the hashtable. Small accumulators can be inlined in the hashtable as an optimization, but this is difficult to do since the Value type is decided at run-time and not at compile-time. The pointers add 16B per entry (on a 64-bit machine).
2. Memory allocator: the key and accumulator are allocated on the heap. Each allocation incurs overhead from the user-space memory allocator, which can be costly if the requested sizes are small. Using an allocator that handles small objects efficiently such as jemalloc [20] reduces the per-key overhead by about 20B compared to the default libc allocator.
3. Hashtable implementation: unoccupied slots in the hashtable waste space in order to limit the load factor of the hashtable for performance reasons. A memory-efficient implementation such as Google's Sparsehash [24] which minimizes the overhead of unoccupied slots and has a per-entry overhead of just 2.67 bits, can be used at the cost of slightly slower inserts. Sparsehash reduces the per-entry overhead by about 16B compared to using the STL `unordered_map`.

As shown in Figure 2b, the per-entry memory overhead for state-of-the-art implementation of hashtables and allocators reflects the high overhead associated with this approach. It is not our intent to optimize this overhead any further in this paper. Instead, we show that hash-based aggregation does not require a hashtable data structure at all.

Compressed Buffer Trees Compressing hashtables is challenging for two reasons:

1. Compressing and decompressing on every access to the hashtable adds unacceptably high overhead.
2. Compression works best on large blocks, but key-value pairs in the hashtable are small. This creates a tension between compression/decompression speed and effectiveness.

The key to effective compression is to be able to perform compression in relatively large chunks while amortizing the cost across multiple update operations. The Compressed Buffer Tree achieves this by taking advantage of the observation that *aggregation can occur lazily*: updates need not be merged immediately, but can be deferred.

Lazy aggregation enables effective compression: multiple accumulators can be buffered together and maintained in compressed form in memory. With eager aggregation, the entire compressed buffer would have to be decompressed if any of the accumulators were required. With lazy aggregation, decompression of the buffer is deferred until we have batched updates to multiple accumulators in the buffer. Thus, the compression costs are now amortized over multiple updates.

How can the system ensure that sufficiently many updates have been batched so that it is worthwhile decompressing a compressed buffer of accumulators? The answer to this question comes from an analogous tradeoff

that exists in external data structures that try to intelligently move data to and from slower external storage.

Consider the implementation of an external binary search tree: immediately inserting an element into the tree requires traversing the tree and performing the insert—a read and a write operation per update, leading to poor performance. The *buffer tree* proposed by Arge [8] instead adds a buffer to each node in the search tree. Inserts are buffered into the root node until it becomes full. The root is then emptied by pushing the values one level down to buffers on the next level. This emptying process is implemented recursively. The important benefit of this solution is that it transforms the original multiple, random I/Os of small updates into I/O involving large buffers.

Buffer trees provide the solution we seek; they enable batching of updates to compressed buffers. The cost of decompressing and compressing buffers in memory is analogous to reading and writing buffers to disk in the setting of external data structures.

Using this insight, we organize compressed buffers into a B-tree keyed by the hashes of the intermediate key. We then use this resulting *Compressed Buffer Tree* for aggregation in MapReduce by performing aggregation only when a buffer is pushed to its children. As a result, the system performs compression and decompression once per buffer flush instead of once per insert operation. The structure trades the O(1) access time of a hashtable (which is not needed in our application) for fast insertion and aggregation into compressed memory.

The CBT also improves memory efficiency through serialization. As a pre-requisite to being able to compress the buffer in memory, inserted values must be serialized. This avoids memory overheads from pointers and further saves memory by using efficient binary encoding. For example, Protobufs [22] serializes integers as *varints* which take one or more bytes allowing smaller values to use fewer bytes.

We describe the design of the Compressed Buffer Tree in Section 3. We then describe the system in which we use and evaluate CBTs, called Minni, our library for MapReduce, in Section 4. We evaluate the CBT and Minni in Section 5 and discuss related work in Section 6.

3 Compressed Buffer Trees

The CBT is based on the buffer tree design for I/O-efficient external data structures [8]. The buffer tree uses an (a,b)-tree with each node augmented by a memory buffer. Inserts and deletes are not performed immediately, but buffered at successive levels of the tree to improve I/O performance. The CBT is an *in-memory* variant of a buffer tree that uses a B-tree and maintains most of the buffers in compressed form for memory-efficiency.

The CBT offers fast, memory-efficient aggregation

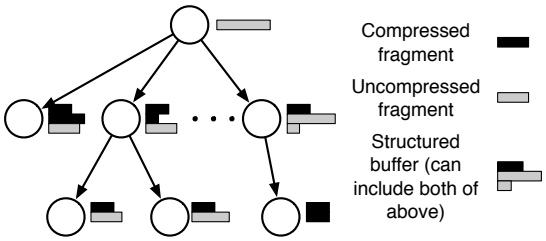


Figure 3: Compressed Buffer Tree

of intermediate key-value pairs and can be used in both the combining and reducing phases of a MapReduce application. The CBT is designed to avoid the overheads associated with hash table-based aggregation through effective use of buffering and compression. The CBT can achieve high throughput, comparable to aggregation using hashtables, despite the additional computational overhead from compression and maintenance operations.

3.1 Partial Aggregation Objects (PAOs)

Our MapReduce system, described in Section 4, aggregates values using a simple data structure called a Partial Aggregation Object (PAO); we introduce PAOs here to help simplify the description of CBTs. The PAO is an abstraction that hides the underlying runtime implementation and provides interfaces for aggregation operations. Before aggregation, each intermediate 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.

3.2 Overview of the basic CBT

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 internal nodes and the leaf nodes are stored in compressed form.

To insert a PAO into the CBT, the PAO is serialized, and 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.

When a buffer reaches some threshold (e.g., half its capacity), it is emptied into the buffers of nodes in the next level. To empty the buffer, the system decompresses the buffer (if not already decompressed), sorts the tuples by hash value, decompresses the buffers of the nodes in the next level and partitions the tuples into those receiving buffers based on the hash value.

If a receiving buffer in the next level fills up as a result of the incoming tuples, the buffer-emptying process is

applied recursively; if the child node is not full (and therefore does not need to spill further), its buffer is compressed. Serializing a large number of PAOs into each buffer substantially increases the effectiveness of compression.

Insertion behaves like a B-tree: a full (again, using some fullness threshold) leaf node splits into two, and the new leaf node is added to the parent of the leaf node. If this causes the number of children of the parent to exceed the maximum allowed, then the parent splits recursively and splitting can propagate up to the root.

As mentioned before, MapReduce aggregation does not require efficient random lookups. The important operations are fast insertion and iterative access to the accumulators after all the PAOs have been inserted and aggregated. For iteration, all the buffers are emptied recursively, flushing all PAOs to the leaf nodes of the CBT. The leaf nodes are then accessed in order and the aggregated PAOs deserialized.

In the naive CBT design described so far, the compression/decompression overhead during insertion is high. We avoid this overhead by restructuring the buffers from simple linear lists to collections of compressed buffer fragments.

3.3 Structured Buffers

Using a single linear buffer for each tree node has several disadvantages. When a buffer is full, the node must spill that data into the buffers of its children. Each child buffer must be decompressed, the data copied, and the buffer re-compressed. Not only does frequent decompression and compression add significant CPU overhead, but this naive design fails to take advantage of the fact that the data copied from the parent is already sorted by hash.

Instead, the buffer at each node holds a set of lists, or buffer fragments. Each copy from parent buffer to child buffer creates a new uncompressed fragment that is appended to the child's buffer. Compressed fragments already in the child remain compressed when a new fragment is added. Only when the buffer is full are all of the fragments decompressed. Since each fragment is already sorted by hash, we can use a fast merge instead of sort. The system ensures that each fragment is large enough so that compression is effective. The structured buffer, shown in Figure 3, both improves throughput and reduces CPU overhead.

3.4 Reducing Insertion Blocking Time

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. Aggregation performance depends significantly on minimizing the amount of time that the insertion thread remains blocked. We describe the techniques that the CBT uses to achieve high throughput by reducing the blocking time of the insertion thread.

Asynchronous operations The first technique we use to reduce the blocking time for the inserter is to make core CBT operations such as compression, sorting/merging and emptying asynchronous. As shown in Figure 4, queues are maintained for each of these operations and nodes are moved between queues after processing. When a node is full, it is inserted into the decompression queue for decompression of all its buffer fragments. It is then moved to the sorting queue, where it is sorted or merged and finally moved to the emptying queue. For example, in Figure 4, if the copy from Node r into Node a causes the latter to become full, Node a is scheduled for decompression (and further merging and emptying) but insertion into Node r resumes once its buffer has been emptied without waiting for Node a or other children to finish emptying.

Prioritized Emptying In the above description, a node will never empty into an already-full child, because nodes are emptied in strictly the same order in which they were queued. However, this strict ordering can increase the blocking time for the insertion thread. For example, if Node d was scheduled to empty before the root r, insertions into the root would be delayed because the root r cannot empty until d has first done so.

We therefore use a level-based prioritization with priority inheritance: the root node has the highest priority and the leaf nodes the lowest. To ensure that full children are emptied before the parent, the parent donates its priority to its full children, if necessary, so that they are emptied first. For example, in Figure 4, Node r cannot be emptied before Node a because the latter is full. The priority of Node a is, therefore, increased in the queue. After Node a is emptied, Node r will empty ahead of Node d owing to its higher priority.

Double-buffered root To further reduce the blocking time of the inserter, we double-buffer the root as shown in Figure 4. This allows insertion to continue while the primary buffer is being emptied. When the secondary buffer is full, the inserter blocks if the primary buffer has not been emptied yet, otherwise the two buffers are swapped and insertion continues.

3.5 Column-Specialized Compression

The CBT borrows the idea of organizing data by columns from column-store databases to enable the application of specialized compression techniques to save memory.

Remember that each buffer fragment in the CBT consists of tuples of the form ⟨`hash, serialized PAO size, serialized PAO`⟩; storing tuples column-wise results in three columns, each of which is compressed separately. For example, in each buffer fragment, the

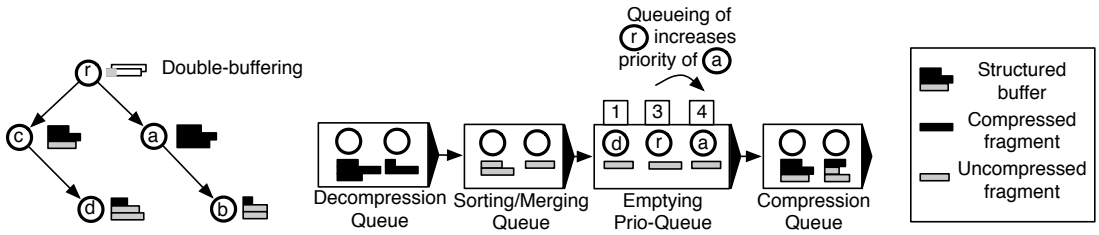


Figure 4: Techniques to reduce inserter blocking: asynchronous operations, prioritized emptying for higher level nodes (Node *r* (root) has highest priority, but if child Node *a* is also queued for emptying, the priority of *a* is increased) and double-buffering of root.

hashes appear in sorted form. Therefore, we use Delta-encoding to compress the column of hashes. Because many of the sizes of the serialized PAOs are similar, we use Run-Length Encoding (RLE) to compress the sizes-of-serialized-PAOs column. The serialized PAOs column is compressed with Snappy [23].

Storing the fields column-wise also improves performance: when the buffer is being sorted or merged, comparisons are made only on the hash values. Storing the hash values of the PAOs in a fragment contiguously allows more hashes to fit in a cache line.

3.6 The CBT API

PAOs are serialized into the root buffer using `insert()`. After all PAOs have been inserted, invoking `flush()` fully aggregates the PAOs by pushing the PAOs to the leaves of the CBT. The aggregated values can then be accessed through an iterator. A bulk version of `insert()` is also provided to minimize function call overhead since the CBT is implemented in a separate shared library.

4 Implementation

We implement the Compressed Buffer Tree as an aggregator in a MapReduce library called Minni. We chose to implement our own MapReduce library for several reasons: First, Hadoop's code structure did not allow us to easily replace their sort-based aggregator. Minni is designed to be flexible and modular. Different components such tokenizers, serializers, etc. can be used to compose different aggregators which can then be selected at runtime. Second, Minni supports multiple cores within a single instance of a mapper or a reducer by structuring aggregators as pipelines. Finally, we wanted to avoid Hadoop's Java-related memory overheads; Minni is written in C++.

4.1 Programming Model

Unlike traditional MapReduce, the intermediate phase in Minni does not operate on raw key-value pairs. Instead, Minni operates on partial aggregation objects or PAOs.

Since traditional MapReduce groups using sorting, it can treat intermediate keys and values as raw strings. However, Minni's aggregator-agnostic PAOs may contain arbitrary intermediate state. The programmer must therefore provide the functions shown in Table 2 to manipulate the PAOs.

Operation	Description
`PAO* create (Token t[])`	Create `PAO` from `Tokens`
`destroy (PAO* p)`	Destroy `p`
`merge (PAO* p1, PAO* p2)`	Merge `p2` into `p1`
`serialize (PAO* p, string* out)`	Serialize `p` to `out`
`deserialize(string* in, PAO* p)`	Deserialize from `in` into `p`

Table 2: The Minni API

Minni reads its input data from a distributed file system (DFS) and creates tokens from the data. Minni includes a set of pre-defined tokenizers such as delimiting tokenizers and file tokenizers. Programmers can also supply their own tokenizers. Table 2 describes the Minni API: The first two functions define how a PAO is created from these tokens and how it is destroyed. The next function defines how to merge a PAO with another that has the same key. The last two functions deal with PAO serialization. Minni supports Protobufs [22] and Boost [1] serialization, or allows the programmer to use custom serialization.

4.2 Minni Architecture

Similar to MapReduce and Hadoop, the Minni master node schedules work on worker nodes. Workers can share data directly over the network or through a distributed filesystem (DFS). Minni currently supports two DFSs: HDFS [4] and KFS [14].

Every Minni job has two parts: a job description (specification) and a dynamically loadable object implementing Minni's API. Based on the job description, the master assigns map and reduce tasks to workers. In Minni, a map task emits a series of locally-aggregated PAOs which are then split using a partitioning function and transferred to reducers over the network.

```
// WordCountPAO inherits from PAO
class WordCountPAO : public PAO {
  string key;
  int count;
};

PAO* WordCountOperations::create(Token toks[]) {
  WordCountPAO *p = new WordCountPAO();
  p->key = tok[0];
  p->count = 1;
  return (PAO*)p;
}
WordCountOperations::destroy(PAO* p) {
  delete p;
}
WordCountOperations::merge(PAO* p1, PAO* p2) {
  WordCountPAO *w1 = (WordCountPAO*)p1;
  WordCountPAO *w2 = (WordCountPAO*)p2;
  w1->count += w2->count;
}
```

Figure 5: Minni implementation of wordcount

4.3 Scalability

There are two options for scaling performance within a single node. Like other MapReduce libraries, Minni allows multiple mappers and reducers to be executed on a single node. Minni also supports pipelining to make use of additional available resources. Minni, therefore, handles the mapper sub-tasks—reading from the DFS, tokenizing the input, hashing the key in the internal hash-table, merging and possibly serializing—as threaded pipeline stages using Intel's Thread Building Blocks [30] `pipeline` construct to do so[1].

Pipelining increases throughput by taking advantage of available CPU and I/O resources with only a modest increase in memory consumption. TBB uses multiple threads to implement different stages in the pipeline and also processes tokens in batches at each stage. Additional memory is required for transfer buffers, but Minni only passes pointers to objects in the transfer buffers to minimize copying costs and memory usage. Running multiple mappers, on the other hand, requires keeping multiple hash-tables or CBTs in memory which use all resources proportionally.

4.4 Example Minni Application

Figure 5 shows an example of `wordcount` implemented using the Minni API listed in Table 2. The `create` function creates a PAO specific to the application and sets the key of this PAO and its value to 1, signifying that so far only 1 token matching the key associated with this PAO has been found. Merging simply involves adding the counts of the two PAOs.

Although different than the normal MapReduce interface of programming only a `map` and `reduce` function, the Minni API is intuitive especially for aggregatable workloads. In addition, the data structure used to store PAOs during aggregation is completely abstracted from the programmer.

[1]This differs from the pipelining used in some implementations of MapReduce [15] where it applies to the pipelining between map, shuffle and reduce stages to obtain early results.

5 Evaluation

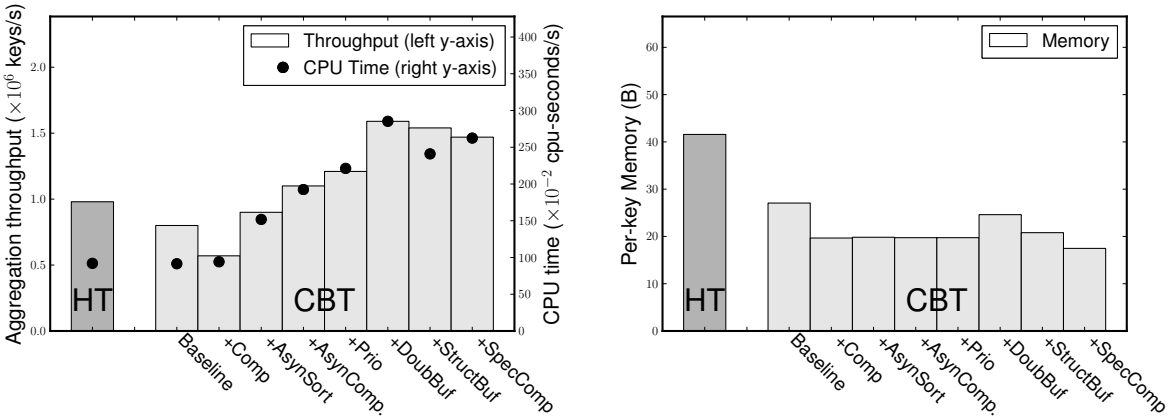
In this section, we evaluate the Compressed Buffer Tree and compare it with alternative data structures for implementing the GroupBy-Aggregate operation. First, we use microbenchmarks to test the effects of workload characteristics and system parameters (§5.1) and, next, we use example applications implemented in Minni (§5.2). To compare the alternatives, we introduce a cost model for the resources based on Amazon EC2 pricing data (§5.3).

Alternatives for hash-based aggregation We compare the CBT against two hashtable implementations. The first, denoted HT, uses Google's `sparse_hash_map` [24], an extremely memory-efficient hashtable implementation; with this comparison, we seek to show that the CBT is often as much as 2× more memory efficient than HT. However, HT does not support concurrent insertion and uses less CPU than CBT making throughput comparisons unfair. Therefore, we also compare against a second *concurrent* hashtable implementation, denoted HT-C, which uses the `concurrent_hash_map` from Intel's Threading Building Blocks [30]. For experiments, we constrain HT-C to use the same amount of CPU as the CBT, allowing for fair throughput comparisons; we seek to show that the performance of the CBT is within an acceptable range of HT-C while being as much as 3× more memory efficient.

Setup The experiments use a 12-core server (two 2.66GHz six-core Intel X5650 processors) running Linux 2.6.32 with 48GB of DDR3 RAM and one 1TB hard disk. Minni uses KFS [14] as the distributed file system to store the input and output datasets; the aggregation state for all experiments is stored completely in memory. Because the focus of this section is to compare data structures for the GroupBy-Aggregate operation, for clarity, we evaluate and report performance for a single GroupBy-Aggregate operation on one node and not the entire job execution time (including shuffle etc.). This operation could represent either a map-side combine operation (to partially aggregate results and lower network bandwidth utilization) or a reduce-side reduction.

5.1 Compressed Buffer Trees

To evaluate the design and benefits of the CBT, we use microbenchmarks consisting of wordcount with a set of synthetic datasets. We define the *aggregatability* of the dataset with respect to an application as the ratio of the size of the dataset to the aggregated size of the dataset. Wordcount's aggregatability is proportional to the average number of occurrences of words in the dataset. The synthetic datasets are represented as $\text{SYN}\langle x, y_{wc}, z\rangle$ where x is the number of unique keys in the dataset, y_{wc}



(a) (b)

Figure 6: Techniques for memory efficiency and high aggregation performance of CBT: a). Aggregation throughput, b). memory efficiency

the aggregatability with respect to wordcount and z the average size of each key. We use randomly generated keys in the synthetic datasets, which is the worst-case for compressibility. The keys are of uniform length and uniformly distributed in the dataset. We consider datasets with variable-length keys and non-uniform distributions in §5.2.

We first evaluate the impact of CBT design features and configuration parameters on aggregation performance and memory consumption. We then evaluate the effects of workload characteristics; and finally, we briefly examine the CBT CPU usage.

5.1.1 Design features

We analyze the performance of the CBT by considering its features in isolation. For each feature, we show its incremental[2] impact on aggregation throughput and memory consumption in Figures 6a and 6b respectively.

Baseline The baseline CBT consists of a buffer tree in memory with no compression enabled, with buffers as simple memory regions with no structure and no asynchronous processing. The basic CBT consumes about 34% less memory compared to HT. 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 PAO into the buffer). Aggregation throughput of the baseline CBT is about 18% less than that of HT.

+Comp Compressing all of the buffers, except the root and a buffer being emptied, with Snappy [23] (in this simplified version of the CBT, no specialized compression is done; instead Snappy is used for the entire buffer) reduces memory use by a further 27%. These are worst-case savings because the randomly generated keys used in the synthetic datasets are not compressible (Snappy uses a dictionary-based algorithm and relies of recurring patterns for effective compression). The cost of the reduction in memory is a loss of aggregation throughput by 28% due to compression occurring synchronously with insertion.

+AsynSort, +AsynComp Compression and sorting consume substantial CPU time, and performing them synchronously hurts insertion performance. By blocking the inserter only when the root buffer is unavailable, as described in §3.4, and compressing and sorting lower level nodes asynchronously, aggregation throughput can be improved by nearly 93% with no impact on memory consumption; the tradeoff is a 104% increase in CPU usage.

+Prio To further reduce inserter blocking time, we use level-based prioritization with priority donation to schedule nodes for emptying, as described in §3.4. This technique improves aggregation throughput by a further 10%, using no additional memory.

+DoubBuf Double-buffering the root (§3.4) allows insertion to continue even when the root is being emptied. Double-buffering increases throughput by nearly 32% at the cost of 24% additional memory. Double buffering nodes other than the root as well as increasing the number of buffers for the root does not increase performance enough to justify the memory use.

+StructBuf Structuring the buffer into multiple buffer fragments avoids decompressing already-compressed fragments while copying data, and allows the use of merging instead of sorting to join buffers. This technique reduces CPU usage by 15%.

[2]Gains/losses are relative to previous version and not to the baseline

+SpecComp Storing the PAO in a column-oriented manner as ⟨`hash,size,serialized PAO`⟩ (§3.5) improves the effectiveness of compression, reducing memory use by a further 16%.

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.

5.1.2 CBT Parameters

Although the CBT avoids many overheads associated with hashtable-based aggregation, it incurs memory overhead when serialized PAOs with the same key occur at multiple levels of the tree as a result of lazy aggregation.

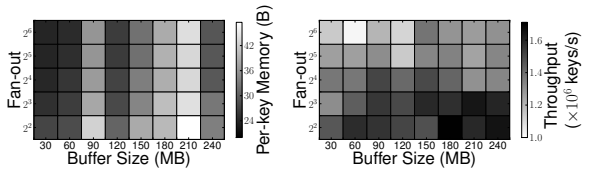
This overhead depends on certain CBT parameters: the node buffer size and the fan-out of the tree. A model for the optimal selection of these parameters is beyond the scope of this paper; instead, we provide some intuition for the dependence of both memory consumption and aggregation performance on these parameters.

Memory Usage The heatmap for memory use in Figure 7a shows that for a given buffer size, the overhead decreases with increasing fan-out (color darkens). This is because a high fan-out decreases the height of the tree and reduces the possible number of occurrence of PAOs with the same key in different levels of the tree. For a given fan-out, the variation of memory use with the buffer size shows an interesting pattern. In general, increasing the buffer size (keeping fan-out fixed) causes increased buffering (or less frequent spilling) leading to higher memory overhead (cf. buffer sizes 150-210MB in Figure 7a). However, larger buffers also result in fewer nodes in the tree which can reduce the height of the tree. This causes a net reduction in memory overhead (cf. buffer sizes: 120, 240MB).

Performance The heatmap for aggregation throughput in Figure 7b shows two trends: First, aggregation performance increases (color darkens) with increasing buffer sizes; this is because larger buffers allow more buffering and less frequent spilling. Second and more prominently, performance decreases with increasing fan-out of the tree; this is because smaller fan-outs increase the height of the tree which leads to a greater number of internal nodes (e.g. with 64 leaf nodes, a fan-out of 4 would require 16+4+1=21 internal nodes, but a fan-out of 16 would require just 4+1=5 internal nodes). A greater number of internal nodes provides more opportunity for buffering, increasing performance.

5.1.3 Workload Properties

Here we consider the three aggregators (CBT, HT and HT-C), and evaluate how three workload properties affect their performance: the number of unique keys, the aggregatability (for wordcount in the following experiments), and the size of the PAO.



(a) (b)

Figure 7: Dependence of per-key memory and aggregation throughput on CBT parameters. For clarity, in this experiment, compression is disabled. The workload used is SYN$\langle 10e8, 10, 8\rangle$; other synthetic datasets were tested and showed similar trends.

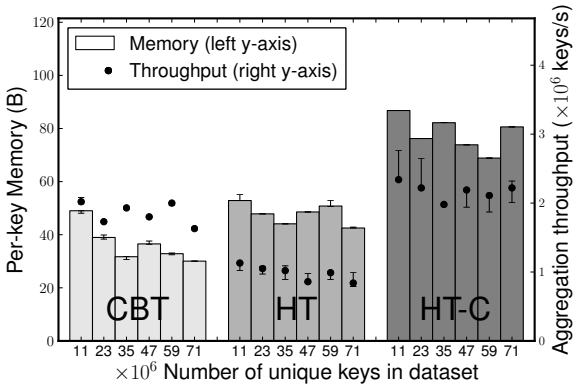


Figure 8: Effect of increasing aggregated data size on memory consumption and performance. The workload used is SYN$\langle x, 50, 8\rangle$ where x is varied.

Aggregated data size The memory use of hash-based aggregation depends on the number of unique keys in the dataset. We use a set of synthetic datasets with an increasing numbers of unique keys while keeping the aggregatability of the dataset constant. Figure 8 shows that the CBT consumes as much as 38% and 65% less memory per key than HT and HT-C respectively, while maintaining a throughput of between 75–97% of HT-C.

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). With a CBT, however, lazy aggregation allows duplicate keys in interior nodes (Figure 9), and so memory increases slightly as the dataset grows. Overall, CBT uses 35% less memory than HT for the values tested.

PAO size Here we grow PAOs by increasing the key size, holding the rest of the parameters constant. Figure 10 shows that the CBT uses less memory than HT,

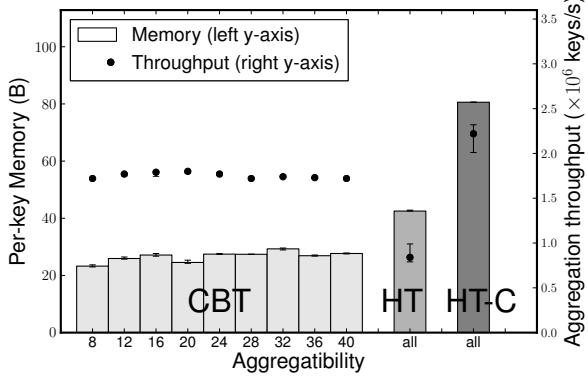


Figure 9: Effect of increasing data aggregatability on CBT performance. The workload used is SYN$\langle 7 * 10^7, x, 8\rangle$ where x is varied.

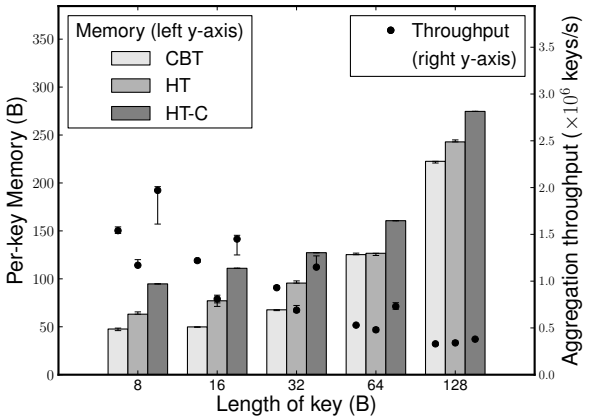


Figure 10: Dependence of performance and memory consumption on key length. The dataset used is SYN$\langle 10^7, 10, x\rangle$ where x is key length.

but the use grows at nearly the same rate as the key size increases. This suggests that when the keys are long and incompressible, the benefit yielded by the CBT decreases. But keys in real applications seldom have these properties. In cases where this is true, this can be replaced by a shorter identifier (e.g. if the key is an image encoding, it can be replaced by its filename or a hash [42]).

5.1.4 CPU Overhead

Figure 6a showed that the benefits in memory efficiency and aggregation throughput cost additional CPU resources. We believe, however, that this tradeoff is reasonable, as memory is increasingly a more limited resource than CPU in big-data environments. Further, we find that compression, merging, sorting and serialization constitute about 65% of the total CPU used during CBT execution. Therefore, there are likely opportunities for more effective algorithms (e.g. more efficient compression algorithm) or for optimized implementation of these algorithms (e.g. with hardware acceleration [44]).

5.2 Example Applications

We next describe three example Minni applications that we implemented to evaluate our CBT-based aggregator. Each application contains a GroupBy-Aggregate operation in the map and reduce phases. Memory consumption and throughput are shown in Figure 11. Unlike the microbenchmarks, the datasets used with these applications have varying key-lengths and distributions of keys.

N-gram Statistics of eBooks An n-gram is a continuous sequence of n items from a given of sequence of text. This application computes n-gram statistics on 30k ebooks downloaded from Project Gutenberg [2]. We calculate the number of instances of 1-, 2- and 3-grams on words in the dataset. N-gram counting is useful in different application such as protein sequencing [52], machine translation [40] and spelling correction [32].

This application tests the ability of the aggregators to handle workloads with large numbers of unique keys. Each PAO contains a key-value pair: the n-gram and a 32-bit count. Merging PAOs simply adds the count values, similar to wordcount.

Image Clustering In this application, we cluster similar images using the perceptual hashes [42] of 80 million images from the MIT Tiny Image dataset [48]. 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 robust enough to handle transformation such as skews, rotation etc. and can be used for duplicate detection.

Each PAO consists of a PH-prefix as key and a list of similar images 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 basic method does not find images that are similar 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.

PageRank This application performs a simplified version of an iterative PageRank computation on the Twitter follower network [33]. The input is an adjacency list of the input graph. For each tuple:

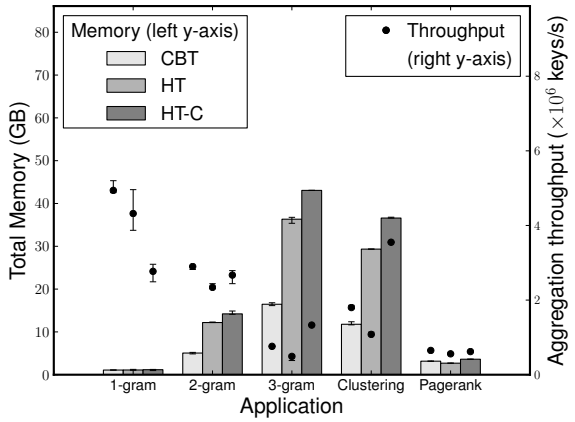


Figure 11: Comparison of CBT with HT and HT-C for example Minni applications

$\langle source \rangle \langle dest_1 \rangle \ldots \langle dest_n \rangle$ in the input, PAOs using *source*, $dest_1$ etc. as keys are created. Each page is given a default PageRank value in the first iteration which is divided among the target pages. A PAO has a string key for the page, a float value for the rank and an optional list of links to pages. PAOs with the same key are merged by adding their associated PageRanks. At the end of each iteration, the aggregated PAOs are serialized to the DFS and become the input for the next iteration.

5.3 Cost model

Tradeoffs between memory consumption, CPU and aggregation throughput appear throughout the system. To decide whether a tradeoff is "worthwhile" requires a cost model for the resources. The model must account for the total cost of operation including purchase and energy costs. Instead of synthesizing a model using such components costs, inspired by an idea in a web log post [38], 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.

Let A be a $m \times 3$ matrix with each row containing the number of units of CPU, memory and storage available in a type of instance for m different kinds of instances. The number of rows is equal to the number of different types of EC2 instances offered (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. Then, solving for x in $A.x = b$ by 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.

Appl.	(x, y)	Hourly rate (¢)			Total cost (¢)		
		HT	HT-C	CBT	HT	HT-C	CBT
1-gram	4, 178	19	39	25	**0.8**	2.6	1
2-gram	103, 7	41	64	35	3.5	4.5	**2.5**
3-gram	462, 2	82	103	47	34.5	15.7	**12.5**
Clustering	286, 5	67	83	33	12.5	4.7	**3.8**
PageRank	36, 40	15	23	18	5.6	7.6	**5.5**

Table 3: Dataset Parameters and Costs using Amazon EC2 cost model: x represents the number of unique keys in the dataset and y represents the average number of occurrences

5.3.1 Analysis

Figure 11 shows the total memory consumption and aggregation throughput for the applications for the different data structures. Table 3 shows the corresponding costs predicted by our model.

Memory Usage CBT is much more memory efficient than HT and HT-C in the case of 2-gram, 3-gram and Clustering applications. In the case of 1-gram and PageRank, the memory use is nearly the same. This is because the number of unique keys in these datasets (Table 3) is small, and the static overhead of CBT dominates the per-key memory. However for 2-grams and 3-grams, which have a high number of unique keys, CBT is more memory efficient than both HT and HT-C.

Throughput CBT uses more CPU than (serial) HT and always yields higher throughput. When compared to HT-C, which uses the same amount of CPU, CBT has higher throughput in cases where there is high aggregatability. High aggregatability implies that on average, keys occur more frequently and this leads to synchronization overhead for HT-C.

Cost Table 3 shows that the CBT is able to reduce the cost of resources through a combination of memory efficiency and high throughput.

6 Related Work

External Memory Data Structures External memory (EM) data structures have been developed primarily to handle very large datasets. Vitter [51] and Arge [7] provide in-depth surveys of EM literature including buffer trees [8], which inspired the CBT. In that work, buffering is used extensively to minimize I/O between in-memory data structures and backing storage. The CBT uses buffering, but not in the same way—buffering paired with lazy aggregation allows compression of the buffers in-memory. The EM research literature [51, 7, 8, 10, 17, 50] considers compression an orthogonal and application-specific technique.

6.1 Compression for Memory Efficiency

Compressed Data Structures Although the CBT applies conventional compression techniques to save memory, the grouping and aggregation occurs on decompressed data. A natural alternative to this approach is to use specialized compressed data structures which allow specific operations to be performed on compressed data. This study of compressed or succinct data structures is advancing rapidly; for example, both Grossi et al. and Välimäki et al. show how compressed suffix trees help in practice for real-world problems by reducing memory use [26, 49]. While we believe this is a promising approach, to our knowledge, no compressed data structure provides a suitable interface to implement a generic GroupBy-Aggregate operation.

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 [12], and Li et al. consider aggregation algorithms in a compressed Multi-dimensional OLAP databases [35]. SILT [36] is a key-value store that introduces a number of techniques, including compression and entropy coding, to build memory-efficient indexes in memory for key-value pairs stored on flash. We believe that the CBT is a promising candidate for implementing aggregation within RDBMS systems.

Memory Compression A number of previous approaches integrate compression into the memory hierarchy in a manner that is transparent to applications [3]. Among software-only approaches, swap compression [47] involves setting aside a partition in the main memory to store evicted pages in compressed form and gain performance by avoiding reads from storage.

6.2 MapReduce and Big Data Analytics

MapReduce [16] has many implementations in different languages [43, 4], with different runtime characteristics [19, 27], and for specific hardware platforms [13, 46]. Its simple programming model of two key API functions—`map` and `reduce`—has been adapted to many problems [5, 6, 39, 41]. Dryad [31] provides a more flexible model of computation over MapReduce. Dryad has been extended to support aggregatable workloads by Yu et al. [53]. Yu et al. describe continuous partial aggregation and associated operations like `merge`, but they do not optimize the storage of PAOs, which we have done with the CBT.

The closest work in spirit to Minni is hash-based MapReduce such as Tenzing [11] or One-Pass MapReduce [34]. Tenzing is a SQL query execution engine built on top of MapReduce [11]. Tenzing requires efficient support for SQL aggregation operations such as `SUM`, `MIN`, etc. and turns to hashtables to implement them. As we have shown, the CBT could replace hashtables in Tenzing for memory efficiency.

Spark [54] attempts to keep as much of a dataset in-memory and notices significant reduction in job times over equivalent MapReduce implementations. Spark is built on top of Mesos [28] and implements "Resilient Distributed Datasets" (RDDs) [54]. RDDs are read-only data structures that are recomputed if pieces are lost. RDDs could be implemented with CBTs, although the aggregation optimizations of the CBT are lost in a read-only setting. Piccolo [45] exposes a shared, distributed, mutable key-value store interface to programmers via Partition Tables. Partition Tables support aggregation of updates to key-value pairs. Piccolo compresses on-disk Partition Tables, but not when they are in memory.

7 Conclusion and Future Work

Experience with important cloud computing infrastructures suggests a growing compute-memory gap: not only do memory costs begin to dominate the overall cost of cloud computing, but the amount of memory available is limited relative to the amount of computation available. As a result, memory-efficient computing techniques may become far more critical.

This paper introduced the Compressed Buffer Tree, a B-Tree-like data structure inspired by external data structures that allows efficient compression of large in-memory append-and-aggregate datasets for memory efficiency and high throughput. The CBT sacrifices $O(1)$ retrieval of the value associated with a key, but works well for workloads that are amenable to lazy aggregation, such as the GroupBy-Aggregate operation found in combining and reducing phases of MapReduce.

This paper also introduced the Minni MapReduce runtime, designed around partial aggregation objects for efficient aggregation. Minni's modular architecture allows drop-in replacement of key components such as the aggregator data structure making it well-suited as an evaluation platform for CBTs.

While the CBT increases the memory efficiency of aggregated data, datasets can be too large to fit in memory. In future work, we plan to address this issue by intelligently moving state from the CBT to fast storage such as SSDs, allowing external aggregation of large datasets.

Although our evaluation and empirical results in this paper focus on MapReduce, the CBT is a general data structure that acts as a drop-in replacement for in-memory aggregators. Yu et al. [53] have shown that the GroupBy-Aggregate operation is crucial for other distributed models such as Dryad and parallel databases. We believe that the CBT will be broadly applicable to these areas.

References

[1] Boost serialization. www.boost.org/libs/serialization.

[2] Project Gutenberg. http://www.guteberg.org/.

[3] 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–. IEEE Computer Society, 2001.

[4] Apache Software Foundation. Hadoop. http://hadoop.apache.org/, 2010.

[5] Apache Software Foundation. Lucene. http://lucene.apache.org/, 2010.

[6] Apache Software Foundation. Mahout. http://mahout.apache.org/, 2010.

[7] L. Arge. External memory data structures. In F. auf der Heide, editor, *Algorithms ESA 2001*, volume 2161 of *Lecture Notes in Computer Science*, pages 1–29. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44676-1_1.

[8] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37 (1):1–24, 2003.

[9] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 117–128. ACM, 2000.

[10] G. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In S. Arnborg and L. Ivansson, editors, *Algorithm Theory SWAT'98*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0054359.

[11] 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.

[12] 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.

[13] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[14] CloudStore. Kosmosfs. http://kosmosfs.sourceforge.net/, 2010.

[15] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1115–1118. ACM, 2010.

[16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[17] R. Dementiev, L. Kettner, and P. Sanders. Standard template library for XXL data sets. In G. Brodal and S. Leonardi, editors, *Algorithms – ESA 2005*, volume 3669 of *Lecture Notes in Computer Science*, pages 640–651. Springer Berlin / Heidelberg, 2005. 10.1007/11561071_57.

[18] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[19] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818. ACM, 2010.

[20] J. Evans. A scalable concurrent malloc(3) implementation for freebsd. *BSDcan*, 2012.

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

[22] Google. Protocol buffers. http://code.google.com/p/protobuf/, .

[23] Google. Snappy. http://code.google.com/p/snappy/, .

[24] Google. Sparsehash. http://code.google.com/p/sparsehash/, .

[25] 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.

[26] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 397–406. ACM, 2000.

[27] Y. Gu and R. L. Grossman. Sector and Sphere: the design and implementation of a high-performance data cloud. *Philosophical Transactions of the Royal*

Society A: Mathematical, Physical and Engineering Sciences, 367(1897):2429–2445, 2009.

[28] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 22–22. USENIX Association, 2011.

[29] U. Hölzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[30] Intel Corporation. Intel threading building blocks. `http://www.threadingbuildingblocks.org/`, 2010.

[31] 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.

[32] 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 - Volume 3*, EMNLP '09, pages 1241–1249. Association for Computational Linguistics, 2009.

[33] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[34] 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.

[35] 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.

[36] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13. ACM, 2011.

[37] 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.

[38] H. Liu. The true cost of an ecu. http://huanliu.wordpress.com/2011/01/24/the-true-cost-of-an-ecu/, 2011.

[39] T. Liu, C. Rosenberg, and H. A. Rowley. Clustering billions of images with large scale nearest neighbor search. In *Proceedings of the Eighth IEEE Workshop on Applications of Computer Vision*, WACV '07, pages 28–. IEEE Computer Society, 2007.

[40] 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.

[41] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and et al. The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data. *Genome Research*, 20(9):1297–1303, 2010.

[42] V. Monga and B. L. Evans. Robust perceptual image hashing using feature points. In *PROC. IEEE CONF. ON IMAGE PROCESSING*, pages 677–680, 2004.

[43] Nokia Research Center. Disco. `http://discoproject.org/`, 2010.

[44] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. Parallel lossless data compression on the gpu. In *Innovative Parallel Computing*, page 9, May 2012.

[45] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14. USENIX Association, 2010.

[46] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos. Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters. *Parallel and Distributed Processing Symposium, International*, 0:1–12, 2009.

[47] L. Rizzo. A very fast algorithm for ram compression. *SIGOPS Oper. Syst. Rev.*, 31(2):36–45, Apr. 1997.

[48] 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.

[49] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. Engineering a compressed suffix tree implementation. *J. Exp. Algorithmics*, 14:2:4.2–2:4.23, Jan.

2010.

[50] D. E. Vengroff. A transparent parallel i/o environment. 1994.

[51] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, June 2001.

[52] J. K. Vries, X. Liu, and I. Bahar. The relationship between n-gram patterns and protein secondary structure. *Proteins: Structure, Function, and Bioinformatics*, 68(4):830–838, 2007.

[53] 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.

[54] 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.