Fast Compaction Algorithms for NoSQL Databases

Mainak Ghosh, Indranil Gupta, Shalmoli Gupta
Department of Computer Science
University of Illinois, Urbana Champaign
Email: {mghosh4, indy, sgupta49}@illinois.edu

Nirman Kumar
Department of Computer Science
University of California, Santa Barbara
Email: nirman@cs.ucsb.edu

***Abstract*—Compaction plays a crucial role in NoSQL systems to ensure a high overall read throughput. In this work, we formally define compaction as an optimization problem that attempts to minimize disk I/O. We prove this problem to be NP-Hard. We then propose a set of algorithms and mathematically analyze upper bounds on worst-case cost. We evaluate the proposed algorithms on real-life workloads. Our results show that our algorithms incur low I/O costs compared to optimal and that a compaction approach using a balanced tree is most preferable.**

1. Introduction

Distributed NoSQL storage systems are being increasingly adopted for a wide variety of applications like online shopping, content management, education, finance etc. Fast read/write performance makes them an attractive option for building efficient back-end systems.

Supporting fast reads and writes simultaneously on a large database can be quite challenging in practice [13], [19]. Since today's workloads are write-heavy, many NoSQL databases [2], [4], [11], [21] choose to optimize writes over reads. Figure 1 shows a typical write path at a server. A given server stores multiple keys. At that server, writes are quickly logged (via appends) to an in-memory data structure called a *memtable*. When the memtable becomes old or large, its contents are sorted by key and flushed to disk. This resulting table, stored on disk, is called an *sstable*.

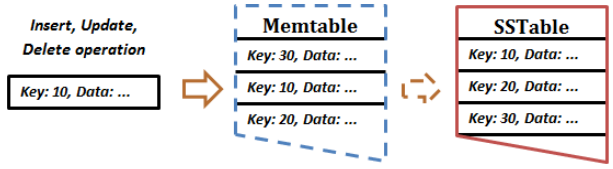


Figure 1: **Schematic representation of a typical write operations. Dashed box represents a memtable. Solid box represents a sstable. Dashed arrow represents flushing of memory to disk.**

This paper was supported in part by the following grants: NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, NSF CCF 1319376, NSF CCF 1217462, NSF CCF 1161495 and a DARPA grant.

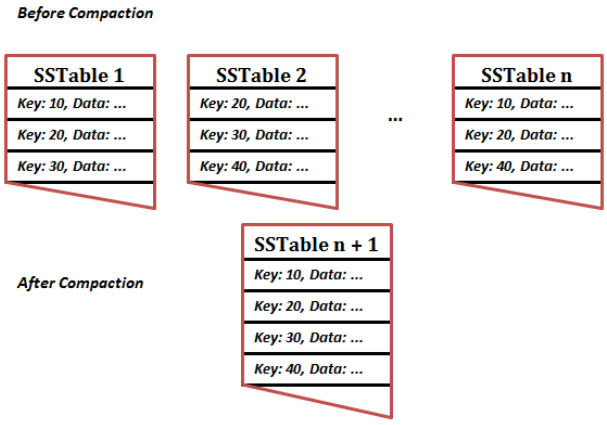


Figure 2: **A compaction operation merge sorts multiple sstables into one sstable.**

Over time, at a server, multiple sstables get generated. Thus, a typical read path may contact multiple sstables, making disk I/O a bottleneck for reads. As a result, reads are slower than writes in NoSQL databases. To make reads faster, NoSQL systems periodically run a *compaction* protocol in the background. Compaction merges multiple sstables into a single sstable by merge-sorting the keys. Figure 2 illustrates an example.

In order to minimally affect normal database CRUD (create, read, update, delete) operations, sstables are merged in iterations. A *compaction strategy* identifies the best candidate sstables to merge during each iteration. To improve read latency, an efficient compaction strategy needs to minimize the compaction running time. Compaction is I/O-bound because sstables need to be read from and written to disk. Thus, to reduce the compaction running time, an optimal compaction strategy should minimize the amount of disk bound data. For the rest of the paper, we will use the term "disk I/O" to refer to this amount of data. We consider the static version of the problem, i.e., the sstables do not change while compaction is in progress.

In this paper, we formulate this compaction strategy as an optimization problem. Given a collection of n sstables, $S_1,\ldots,S_n$, which contain keys from a set, U, a compaction strategy creates a *merge schedule*. A merge schedule defines

a sequence of sstable merge operations that reduces the initial n sstables into one final sstable containing all keys in U. Each merge operation reads atmost k sstables from disk and writes the merged sstable back to disk (k is fixed and given). The total disk I/O cost for a single merge operation is thus equal to the sum of the size of the input sstables (that are read from disk) and the merged sstable (that is written to disk). The total cost of a merge schedule is the sum of the cost over all the merge operations in the schedule. An optimal merge schedule minimizes this cost.

Our Contribution. In this paper, we thoroughly study the compaction problem from a theoretical perspective. We formalize the compaction problem as an optimization problem. We further show a generalization of the problem which can model a wide class of compaction cost functions. Our contributions are as follows:

- We prove that the optimization problem is NP-hard (Section 3).
- We propose a set of greedy algorithms for the compaction problem with provable approximation guarantees (Section 4).
- We quantitatively evaluate the greedy algorithms with real-life workloads using our implementation (Section 5).

Related Work. Bigtable [15] was among the first systems to implement compaction. This system merges sstables when the number of sstables reaches a pre-defined threshold. They do not optimize for disk I/O. When the workload is read-heavy, running compaction over multiple iterations is slow in achieving the desired read throughput. To solve this, Level-based compaction [7], [9] merges every insert, updates and deletes instead. Thus they optimize for reads by sacrificing writes. NoSQL databases like Cassandra [1] and Riak [10] implement both these strategies [8], [12]. Cassandra's Size-Tiered compaction strategy [12], inspired from Google's Bigtable, merges sstables of equal size. This approach bears resemblance to our SMALLESTINPUT heuristic defined in Section 4. For data which becomes immutable over time, such as logs, specific compaction strategies have been proposed [3], [25] which look to prioritize compaction for recent data as they are read more often. The goals of these strategies are orthogonal to ours. Outside ours, Mathieu et. al. [24] have also theoretically analyzed the compaction problem. In their cost function, they optimize for CPU load which is proportional to the sum of the cardinality of the sstables they merge in a iteration. Their merge schedule determines the number of sstables to merge in a iteration. Even though we optimize for different resources, it would be worthwhile to compare our strategies. We plan to do this in the future.

2. Problem Definition

Consider the compaction problem on n sstables for the case where $k = 2$, i.e., in each iteration, 2 sstables are merged into one. As we discussed in Section 1, an sstable consists of multiple entries, where each entry has a key and associated values. When 2 sstables are merged, the new sstable is created which contains only one entry per key present in either of the two base sstables. To give a theoretical formulation for the problem, we assume that: 1) all key-value pairs are of the same size, and 2) the value is comprehensive, i.e., contains all columns associated with a key. This makes the size of an sstable proportional to the number of keys it contains. Thus an sstable can be considered as a set of keys and a merge operation on sstables performs simple union of sets (where each sstable is a set). With this intuition, we can model the compaction problem for $k = 2$ as the following optimization problem.

Given a ground set $U = \{e_1, \dots e_m\}$ of m elements, and a collection of n sets (sstables), $A_1, \dots, A_n$ where each $A_i \subseteq U$, the goal is to come up with an optimal merge schedule. A merge schedule is an ordered sequence of set union operations that reduces the initial collection of sets to a single set. Consider the collection of sets, initially $A_1, \dots, A_n$. At each step we merge two sets (*input sets*) in the collection, where a merge operation consists of removing the two sets from the collection, and adding their union (*output set*) to the collection. The cost of a single merge operation is equal to the sum of the sizes of the two input sets plus the size of the output set in that step. With n initial sets there need to be $(n-1)$ merge operations in a merge schedule, and the total cost of the merge schedule is the sum of the costs of its constituent merge operations.

Observe that any merge schedule with $k = 2$ creates a full[1] binary tree T with n leaves. Each leaf node in the tree corresponds to some initial set A_i, each internal node corresponds to the union of the sets at the two children, and the root node corresponds to the final set. We assume that the leaves of T are numbered $1, \dots, n$ in some canonical fashion, for example using an in-order traversal. Thus a merge schedule can be viewed as a full binary tree T with n leaves, and a permutation $\pi : [n] \to [n]$ that assigns set A_i (for $1 \leq i \leq n$), to the leaf numbered $\pi(i)$. We call this the *merge tree*. Once the merge tree is fixed, the sets corresponding to the internal nodes are also well defined. We label each node by the set corresponding to that node. By doing a bottom-up traversal one can label each internal node. Let ν be an internal node of such a tree and A_ν be its label. For simplicity, we will use the term *size of node* ν, to denote the cardinality of A_ν.

In our cost function the size of a leaf node or the root node is counted only once. However, for an internal node (non-leaf, non-root node) it is counted twice, once as input, and once as output. Let V' be the set of internal nodes. Formally, we define the cost of the merge schedule as:

$$\mathsf{cost}_{actual}(T, \pi, A_1, \dots, A_n) = \sum_{\nu \in V'} 2\,|A_\nu| + \sum_{i=1}^{n} |A_i| + |A_{root}|$$

Then, the problem of computing the optimal merge schedule is to create a full binary tree T with n leaves, and an assignment π of sets to the leaf nodes such that

1. A binary tree is *full* if every non-leaf node has two children

$\mathsf{cost}_{actual}(T, \pi, A_1, \ldots, A_n)$ is minimized. This cost function can be further simplified as follows:

$$\mathsf{cost}(T, \pi, A_1, \ldots, A_n) = \sum_{\nu \in T} |A_\nu| \tag{2.1}$$

The optimization problems over the two cost functions are equivalent because the size of the leaf nodes, and the root node is constant for a given instance. Further, an α-approximation for $\mathsf{cost}(T, \pi, A_1, \ldots, A_n)$ immediately gives a $2 \cdot \alpha$-approximation for $\mathsf{cost}_{actual}(T, \pi, A_1, \ldots, A_n)$. For ease of exposition, we use the simplified cost function in equation (2.1) for all the theoretical analysis presented in this paper. We call this optimization problem as the BINARYMERGING problem. We denote the optimal cost by $\mathsf{opt}_s(A_1, \ldots, A_n)$.

A Reformulation of the Cost. A useful way to reformulate the cost function $\mathsf{cost}(T, \pi, A_1, \ldots, A_n)$ is to count the cost per element of U. Since the cost of each internal node is just the size of the set that labels the node, we can say that the cost receives a contribution of 1 from an element at a node if it appears in the set labeling that node. The cost can now be reformulated in the following manner. For a given element $x \in U$, let $T(x)$ denote the *minimal subtree* of T that spans all the nodes ν in T whose label sets $\pi(\nu)$ contain x and the root node. Let $|T(x)|$ denote the number of edges in $T(x)$. Then we have that:

$$\mathsf{cost}(T, \pi, A_1, \ldots, A_n) = \sum_{x \in U} (|T(x)| + 1). \tag{2.2}$$

Relation to the problem of Huffman Coding. We can view the problem of Huffman coding as a special case of the BINARYMERGING problem. Suppose we have n disjoint sets $A_1, \ldots, A_n$ with sizes $|A_i| = p_i$. We can see that, using the full binary tree view and the reformulated cost in equation (2.2), the cost function is the same as the problem of an optimal prefix free code on n characters with frequencies $p_1, \ldots, p_n$.

Generalization of BINARYMERGING. As we saw, BINARYMERGING models a special case of the compaction problem where in each iteration 2 sstables are merged. However in the more general case, one may merge atmost k sstables in each iteration. To model this, we introduce a natural generalization of the BINARYMERGING problem called the K-WAYMERGING problem. Formally, given a collection of n sets, $A_1, \ldots, A_n$, covering a groundset U of m elements, and a parameter k, the goal is to merge the sets into a single set, such that at each step: 1) atmost k sets are merged and 2) the merge cost is minimized. The cost function is defined similar to BINARYMERGING.

Extension to Submodular Cost Function. In BINARYMERGING, we defined the cost of a merge operation as the cardinality of the set created in that merge step. However, in real-world situation the merge cost can be more complex. Consider two such examples: 1) when two sstables are merged, the cost of the merge not only involves the size of the new sstable but also a constant cost may be involved with initializing a new sstable. 2) keys can have a non-negative weight (e.g., size of an entry corresponding to that key), and the merge cost of two sstables can be defined as the sum of the weights of the keys in the resultant merged sstable. Both these cost functions (and also the one used in BINARYMERGING), fall under a very important class of functions called *monotone submodular function*. Formally such a function is defined as follows:

Consider a set function $f : 2^U \to \mathbb{R}$, which maps subsets $S \subseteq U$ of a finite ground set U to real numbers. f is called monotone if $f(S) \leq f(T)$ whenever $S \subseteq T$. f is called submodular if for any $S, T \subseteq U$, we have $f(S \cup T) + f(S \cap T) \leq f(S) + f(T)$ [22].

We extend the BINARYMERGING problem to use submodular merge cost function. We call it the SUBMODULARMERGING problem: given a monotone submodular function f on the groundset U, and n initial sets $A_1, \ldots A_n$ over U, the goal is to merge them into a single set such that the total merge cost is minimized. If two sets $X, Y \subseteq U$ are merged, then the cost is given by $f(X \cup Y)$. Note if the function f is, $f(X) = |X|$ for any $X \subseteq U$, it gives us the BINARYMERGING problem. The approximation results we present in this paper extends to this more general SUBMODULARMERGING problem also.

Our Results. In this paper, we primarily focus on the BINARYMERGING problem. The main theoretical results of this paper are as follows:

- We prove that the BINARYMERGING problem is NP-hard (Section 3). Since the K-WAYMERGING, and the SUBMODULARMERGING are more general problems, their hardness follows immediately.
- We show that the BINARYMERGING problem can be polynomial time approximated to $\min\{O(\log n), f)\}$, where n is the number of initial sets, and f is the maximum number of sets in which any element appears (Section 4). The results extend for K-WAYMERGING and SUBMODULARMERGING.

3. BINARYMERGING is NP-hard

In this section, we provide an intuitive overview of the NP-hardness proof of the BINARYMERGING problem. The formal detailed proof is given in Appendix A.

The BINARYMERGING problem offers combinatorial choices along two dimensions: first, in the choice of the full binary tree T, and second, in the labeling function π that assigns the sets to the leaves of T. Intuitively, this should make the problem somewhat harder compared to the case of fewer choices. However, surprisingly it is more challenging to prove hardness with more choices. When the tree is fixed, we call the problem the OPT-TREE-ASSIGN problem (see Appendix A.2 for the definition).

Suppose the tree T is fixed to be the caterpillar tree T_n: such a tree has n leaf nodes and height $(n-1)$. It can be defined recursively as follows. For $n = 2$, T_2 is the balanced

tree with 2 leaf nodes. For $n > 2$, T_n is defined by making the left leaf of T_2 to be the root node of T_{n-1}. Figure 3 shows a caterpillar T_n.

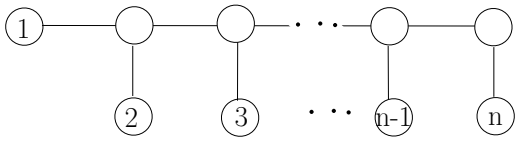


Figure 3: **A caterpillar tree with n leaf nodes (T_n).**

If this tree is fixed as the merge tree, the problem is to choose an optimal labeling function π. We can show that this problem is NP-hard, by a reduction from the precedence constrained scheduling problem, see [16]. Unfortunately, we cannot really use this result to prove the hardness for the BINARYMERGING problem, for reasons detailed below.

To prove that the BINARYMERGING problem is NP-hard, our general strategy is to force the tree T to be a fixed tree and to leave the choice to the labeling function. Intuitively, this should help because several well-known ordering problems are NP-hard. In order to fix the tree we modify the sets so that we can force the optimal tree to have a given structure, and at the same time, the solution to the given instance can be inferred from the new instance.

One way to gain some control over the tree T is as follows. Suppose instead of sets A_i we replace them by $A_i \cup B_i$ where B_i are some large sets. If we choose the sets B_i to be all disjoint from each other and the sets A_i, the tree structure starts to be dominated by the solution for the sets B_i. In other words, the sets A_i, appear to be noise compared to the sets B_i. By carefully choosing the sizes of the sets B_i we can force *any* full binary tree to be T. It would seem that the reduction should now be easy as we can force the caterpillar tree and thus achieve our hardness result. However, there is an additional challenge. As we choose the sets B_i, not only the structure but also the labeling starts to be fixed in an optimal solution for the sets $A_i \cup B_i$. In particular, for the caterpillar tree, the ordering is completely fixed, (although we do not prove this here). Fortunately, if the merge tree is forced to be the completely balanced tree $\mathcal{T}$, we still have *complete choice* in the labeling function. Thus, our strategy in proving the hardness of the BINARYMERGING problem proceeds as follows:

(A) We show that if the tree T is fixed to be the complete binary tree $\mathcal{T}$, then indeed the OPT-TREE-ASSIGN problem is NP-hard. This is done by a reduction from the SIMPLE DATA ARRANGEMENT problem, introduced by Luczak and Noble [23]. We reproduce the definition of this problem, as well as provide the reduction, in Appendix A.1.

(B) In Appendix A.3, we show how to force the tree T to be the complete binary tree $\mathcal{T}$. Intuitively, if the BINARYMERGING problem is run on sets B_i, where B_i are all disjoint and the same size then the merge tree must be $\mathcal{T}$. This is not too hard to believe owing to symmetry. Recall however that the input sets to our new instance of the BINARYMERGING problem are $A_i \cup B_i$ for $i = 1, \ldots, n$. In order to prove that the optimal tree still remains $\mathcal{T}$ we show that if the tree were not $\mathcal{T}$, the cost increment because of the sets B_i would offset any conceivable gain coming from the sets A_i (due to a different tree). To achieve this we make use of a bound on the sum of all root to leaf path lengths, see Lemma A.2, and several small observations that split the total cost of the instance with sets $A_i \cup B_i$ into that of the instances with only sets A_i and the size of B_i (recall that all of them have the same size). Putting all this together, we finally have our desired reduction.

4. Greedy Heuristics for BINARYMERGING

In this section, we present and analyze four greedy heuristics that approximate an optimal merge schedule. We start by giving a lower bound on the cost of the optimal merge schedule. Later, we will use this lower bound to prove the approximation ratio for our greedy heuristics.

4.1. A Lower bound on Optimal Cost

We know (refer Section 2) that $\mathsf{OPT} = \mathsf{opt}_s(A_1, \ldots, A_n)$ is the cost of the optimal merge schedule. Let, Cost denote the cost of the merge schedule returned by our algorithm. To give an α-approximate algorithm, we need to show that $\mathsf{Cost} \leq \alpha \cdot \mathsf{OPT}$. Since OPT is not known, we instead show that $\mathsf{Cost} \leq \beta \cdot \mathsf{L_{OPT}}$, where $\mathsf{L_{OPT}}$ is a lower bound on OPT. This gives an approximation bound with respect to OPT itself. Observe that $\mathsf{OPT} \geq \sum_{i=1}^{n} |A_i|$. This follows immediately from the cost function (equation (2.2)), since the cost function size of each node in the merge tree is considered once and sum of the sizes of leaf nodes is $\sum_{i=1}^{n} |A_i|$. Henceforth, we use $\sum_{i=1}^{n} |A_i|$ as $\mathsf{L_{OPT}}$.

4.2. Generic Framework for Greedy Algorithm

The four greedy algorithms we present in this section are special cases of a general approach, which we call the GREEDYBINARYMERGING algorithm. The algorithm proceeds as follows: at any time it maintains a collection of sets C, initialized to the n input sets $A_1, \ldots A_n$. The algorithm runs iteratively. In each iteration, it calls the subroutine CHOOSETWOSETS, to choose greedily two sets from the collection C to merge. This subroutine implements the specific greedy heuristic. The two chosen sets are removed from the collection and replaced by their union i.e., the merged set. After $(n-1)$ iterations only 1 set remains in the collection and the algorithm terminates. Details are formally presented in Algorithm 1.

4.3. Heuristics

We present 4 heuristics for the CHOOSETWOSETS subroutine in the GREEDYBINARYMERGING algorithm.

```
Algorithm GreedyBinaryMerging(A_1, ... A_n)
    C ← {A_1, ..., A_n};
    for i = 1, ..., n − 1 do
        S_1, S_2 ← ChooseTwoSets(C);
        C ← C \ {S_1, S_2};
        C ← C ∪ {S_1 ∪ S_2};
    end
```

Algorithm 1: Generic greedy algorithm.

We show that three of these heuristics are $O(\log n)$-approximations. To explain the algorithms we will use the following working example:

Working Example. We are given as input 5 sets: $A_1 = \{1, 2, 3, 5\}$, $A_2 = \{1, 2, 3, 4\}$, $A_3 = \{3, 4, 5\}$, $A_4 = \{6, 7, 8\}$, $A_5 = \{7, 8, 9\}$. The goal is to merge them into a single set such that the merge cost as defined in Section 2 is minimized.

4.3.1. BALANCETREE (BT) Heuristic. Assume for simplicity that n is a power of 2. One natural heuristic for the problem is to merge in a way such that the underlying merge tree is a complete binary tree. This can be easily done as follows: the input sets form the leaf nodes or level 1 nodes. The n leaf nodes are arbitrarily divided into $n/2$ pairs. The paired sets are merged to get the level 2 nodes. In general, the level i nodes are arbitrarily divided into $n/2^i$ pairs. Each pair is merged i.e., the corresponding sets are merged to get $n/2^i$ nodes in the $i+1^{th}$ level. This builds a complete binary tree of height $\log n$.

However, when n is not a power of 2, to create a merge tree of height $\lceil \log n \rceil$ involves a little more technicality. To do this, annotate each set with its level number l, and let $minL$ be the minimum level number across all sets at any point of time. Initially, all the sets are marked with $l = 1$. In each iteration, we choose two sets whose level number is $minL$, merge these sets, and assign the new merged set the level $(minL + 1)$. If only 1 set exists with level number equal to $minL$, we increment its l by 1 and retry the process. Figure 4 shows the merge schedule obtained using this heuristic on our working example.

Lemma 4.1. *Consider an instance $A_1, \ldots, A_n$ of the* BINARYMERGING *problem.* BALANCETREE *heuristic, gives a $(\lceil \log n \rceil + 1)$-approximation.*

Proof. Let T be the merge tree constructed. By our level-based construction, $height(T) = \lceil \log n \rceil$. Let C^l denote the collection of sets at level l. Now observe that each set in C^l is either the union of some initial sets, or is an initial set by itself. Also, each initial set participates in the construction of atmost 1 set in C^l. This implies that:

$$\sum_{S \in C^l} |S| \leq \sum_{i=1}^{n} |A_i| = \mathsf{L_{OPT}} \leq \mathsf{OPT}$$

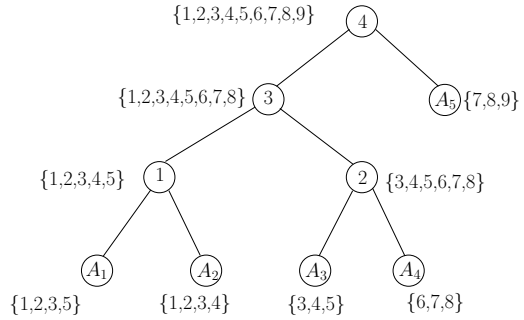


Figure 4: **Merge schedule using BALANCETREE heuristic. The label inside the leaf nodes denotes the corresponding set. The label inside internal nodes denote the iteration in which the merge happened. The sets corresponding to each node is shown outside the node. Cost of the merge = 45.**

Therefore,

$$\mathsf{Cost} = \sum_{l=1}^{\lceil \log n \rceil + 1} \sum_{S \in C^l} |S| \leq (\lceil \log n \rceil + 1) \cdot \mathsf{OPT}$$

■

Lemma 4.2. *The approximation bound proved for* BALANCETREE *in Lemma 4.1 heuristic is tight.*

Proof. We show an example where the merge cost obtained by using BALANCETREE heuristic is $\Omega(\log n) \cdot \mathsf{OPT}$. Consider n initial sets where n is a power of 2. The sets are $A_1 = \{1\}$, $A_2 = \{1\}, \ldots, A_{n-1} = \{1\}$, $A_n = \{1, 2, 3, \ldots n\}$, i.e., we have $(n - 1)$ identical sets which contain just the element 1, and one set which has n elements. An optimal merge schedule is the *left-to-right merge*, i.e., it starts by merging A_1 and A_2 to get the set $A_1 \cup A_2$, then merges $A_1 \cup A_2$ with A_3 to get $A_1 \cup A_2 \cup A_3$ and so on. The cost of this merge is $(4n - 3)$. However the BALANCETREE heuristic creates a complete binary tree of height $\log n$, and the large n size set $\{1, 2, \ldots, n\}$ appears in every level. Thus the cost will be atleast $n \cdot (\log n + 1)$. This lower bounds the approximation ratio of BALANCETREE heuristic to $\Omega(\log n)$. ■

4.3.2. SMALLESTINPUT (SI) Heuristic. This heuristic selects in each iteration, those two sets in the collection that have the smallest cardinality. The intuitive reason behind this approach is to defer till later the larger sets and thus, reduce the recurring effect on cost. Figure 5 shows the merge tree we obtain when we run the greedy algorithm with SMALLESTINPUT heuristic on our working example.

4.3.3. SMALLESTOUTPUT (SO) Heuristic. In each iteration, this heuristic chooses those two sets in the collection whose union has the least cardinality. The intuition behind this approach is similar to SI. In particular, when the sets $A_1, \ldots, A_n$ are all disjoint, these two heuristics lead to the

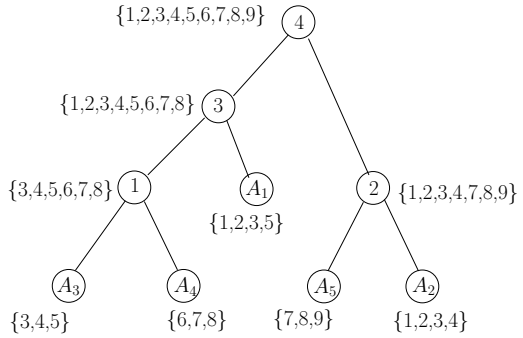


Figure 5: **Merge schedule using SMALLESTINPUT heuristic. Initially the smallest sets are $A_3,\ A_4,\ A_5$. The algorithm arbitrarily choses A_3 and A_4 to merge, creating node 1 with corresponding set $\{3,4,5,6,7,8\}$. Next the algorithm proceeds with merging A_1 and A_2 as they are the current smallest sets in collection, and so on. Cost of the merge = 47.**

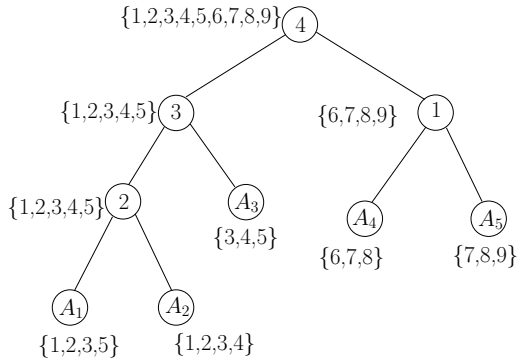


Figure 6: **Merge schedule using SMALLESTOUTPUT heuristic. Initially the smallest output set is obtained by merging sets $A_4,\ A_5$. In first iteration $A_4,\ A_5$ is merged to get the new set $\{6,7,8,9\}$. Next the algorithm chooses $A_1,\ A_2$ to merge as they create the smallest output of size 4, and so on. Cost of the merge = 40.**

same algorithm. Figure 6 depicts the merge tree we obtain when executed on our working example.

Lemma 4.3. *Given n disjoint sets $A_1, \ldots, A_n$, the* BINARYMERGING *problem can be solved optimally using* SMALLESTINPUT *(or* SMALLESTOUTPUT*) heuristics.*

Proof. As we remarked in Section 2 that for this special case, the BINARYMERGING problem reduces to the Huffman coding problem, and as is well known, the above greedy heuristic is indeed the optimal greedy algorithm for prefix free coding [20]. ■

Lemma 4.4. *Consider an instance $A_1, \ldots, A_n$ of the* BINARYMERGING *problem. Both the* SMALLESTINPUT *and* SMALLESTOUTPUT *heuristics, give $O(\log n)$ approximate solutions.*

Proof. Let $A_1^j, , \ldots, A_{n-j}^j$, be the sets left after the j^{th} iteration of the algorithm. Now observe that each A_i^j is either the union of some initial sets, or is an initial set itself. Further each initial set contributes to exactly 1 of the A_i^j's. This implies that:

$$\sum_{i=1}^{n-j} \left|A_i^j\right| \leq \sum_{i=1}^{n} |A_i| = \mathsf{L_{OPT}} \leq \mathsf{OPT}$$

Without loss of generality, let us assume that after j iterations, A_1^j and A_2^j are the two smallest cardinality sets left. We can show that (see Lemma B.1):

$$\left|A_1^j \cup A_2^j\right| \leq \left|A_1^j\right| + \left|A_2^j\right| \leq \frac{2}{n-j} \sum_{i=1}^{n-j} \left|A_i^j\right|$$

If the greedy algorithm uses the SMALLESTINPUT heuristic, then in the $(j+1)^{th}$ iteration, sets A_1^j, A_2^j will be chosen to be merged. In case of the SMALLESTOUTPUT heuristic, we choose the two sets that give the smallest output set. Let C_{j+1} be the output set created in the $(j+1)^{th}$ iteration. Combining the above we can say that:

$$C_{j+1} \leq \left|A_1^j \cup A_2^j\right| \leq \frac{2}{n-j} \cdot \mathsf{OPT}$$

Thus, for either of the greedy strategies, SMALLESTINPUT and SMALLESTOUTPUT, the total cost is:

$$\begin{aligned}\mathsf{Cost} &\leq \sum_{i=1}^{n} |A_i| + \sum_{j=1}^{n-1} |C_j| \leq \mathsf{OPT} + \sum_{j=1}^{n-1} \frac{2}{n-j+1} \cdot \mathsf{OPT} \\ &\leq (2H_n + 1) \cdot \mathsf{OPT} \quad [H_n \text{ is the } n^{th} \text{ harmonic number}]\end{aligned}$$

■

Lemma 4.5. *The greedy analysis is tight with respect to the lower bound for optimal ($\mathsf{L_{OPT}}$).*

Proof. We show an example where the ratio of the cost of merge obtained by using SMALLESTINPUT or SMALLESTOUTPUT heuristic and $\mathsf{L_{OPT}}$ is $\log n$. Consider n initial sets where n is a power of 2. The sets are $A_1 = \{1\}, \ldots, A_i = \{i\}, \ldots, A_n = \{n\}$, i.e., each set is of size 1 and they are disjoint. The lower bound we used for the greedy analysis is $\mathsf{L_{OPT}} = \sum_{i=1}^{n} |A_i| = n$. Both the heuristics, SMALLESTINPUT and SMALLESTOUTPUT, creates a complete binary tree of height $\log n$. Since the initial sets are disjoint, the collection of sets in each level is also disjoint and the total size of the sets in each level is n. Thus the total merge cost is $n \cdot \log n = \log n \cdot \mathsf{L_{OPT}}$. ■

Remark. Lemma 4.5 gives a lower bound with respect to $\mathsf{L_{OPT}}$, and not OPT. It suggests that the approximation ratio cannot be improved unless the lower bound ($\mathsf{L_{OPT}}$) is refined. Finding a bad example with respect to OPT is an open problem.

4.3.4. LARGESTMATCH Heuristic. In each iteration, this approach chooses those two sets that have largest intersection [6]. However, the worst case performance bound for this heuristic can be arbitrarily bad. It can be shown that the approximation bound for this algorithm is $\Omega(n)$. Consider a collection of n sets, where set $A_i = \{1, 2, \ldots, 2^{i-1}\}$, for all $i \in [n]$. The optimal way of merging is left-to-right merge. The cost of this merge is $1+2\cdot(2+4+\ldots 2^{n-1}) = 2^{n+1}-3$. However, the LARGESTMATCH heuristic will always choose $\{1, 2, \ldots, 2^{n-1}\}$ as one of the sets in each iteration as it has largest intersection with any other set. Thus the cost will be $2^{n-1}\cdot(n-1)$. This shows a gap of $\Omega(n)$ between the optimal cost and LARGESTMATCH heuristic.

4.4. An f-approximation for BINARYMERGING

For each element x in U, let f_x denote the number of initial sets to which x belongs, i.e., the f_x is frequency of x in the initial sets. Let $f = \max_{x \in U} f_x$ denote the maximum frequency across all elements. We present an f-approximation algorithm for BINARYMERGING. If f is small, i.e., the elements do not belong to a large number of sets, then this algorithm gives stronger approximation bound than the preceding algorithms. The algorithm is shown in Algorithm 2 and proceeds as follows: we create a dummy set A'_i corresponding to each initial set A_i. These dummy sets are obtained by replacing each element in a set by a tuple, which consists of the element and the set number. Note that dummy sets created in this manner are disjoint. We run the GREEDYBINARYMERGING on the sets $A'_1, \ldots A'_n$ using SMALLESTINPUT (or SMALLESTOUTPUT) heuristic to obtain the tree T' and leaf assignment function π'. Finally, we use the same T', and π' to merge the initial sets. The intuition behind the algorithm is as follows: once the sets are disjoint our algorithms perform optimally and the resultant tree can be used as a guideline for merging.

```
1 Algorithm FREQBINARYMERGING(A_1, ... A_n)
2     Corresponding to each set A_i create a new set
      A'_i, where A'_i = {(x, i) : x ∈ A_i};
3     Run GREEDYBINARYMERGING on
      A'_1, A'_2, ..., A'_n with SMALLESTINPUT heuristic;
4     Let T' be the tree and π' be the leaf assignment;
5     Merge A_1, ... A_n using T', and π'
```

Algorithm 2: f-approx for BINARYMERGING.

Lemma 4.6. *Algorithm 2 is an f-approximation algorithm for* BINARYMERGING.

Proof. Let OPT' be the optimal merge cost for the instance $A'_1, \ldots A'_n$. Let Cost' be the cost of the greedy solution. The sets $A'_1, \ldots A'_n$ are disjoint by construction. By Lemma 4.3, the SMALLESTINPUT (or SMALLESTOUTPUT) heuristic gives the optimal solution in this case. This implies $\mathsf{OPT}' = \mathsf{Cost}'$. Let ν be any node in T'. Le A'_ν be its label for the instance $A'_1, \ldots A'_n$ and A_ν be its label for the instance $A_1, \ldots A_n$. A_ν is union of some initial sets, and A'_ν is the union of corresponding modified initial sets which are disjoint. It follows that $|A_\nu| \leq |A'_\nu|$. Summing we get, $\mathsf{Cost} \leq \mathsf{Cost}' = \mathsf{OPT}'$.

For the instance $A_1, \ldots A_n$, let T_{OPT} be the optimal tree, and π_{OPT} be the leaf assignment . Now if $A'_1, \ldots A'_n$ was merged using T_{OPT} and π_{OPT}, then the cost of the merge will be at most $f\cdot\mathsf{OPT}$. This follows from the fact that size of each node in the new tree is atmost f times the size of the corresponding node in the optimal tree, as each set can contain atmost f renamed copies of the same element. Since T_{OPT} and π_{OPT} are not optimal for $A'_1, \ldots A'_n$ the resulting merge cost is atleast OPT' i.e., $\mathsf{OPT}' \leq f \cdot \mathsf{OPT}$.

Combining we get, $\mathsf{Cost} \leq f \cdot \mathsf{OPT}$. ■

5. Simulation Results

In this section, we evaluate our greedy strategies from Section 4 in practice. Our experiments answer the following questions:

- Which compaction strategy should be used in practice, given real-life workloads?
- How close is a given compaction strategy to optimal?
- How effective is the cost function in modeling running time for compaction?

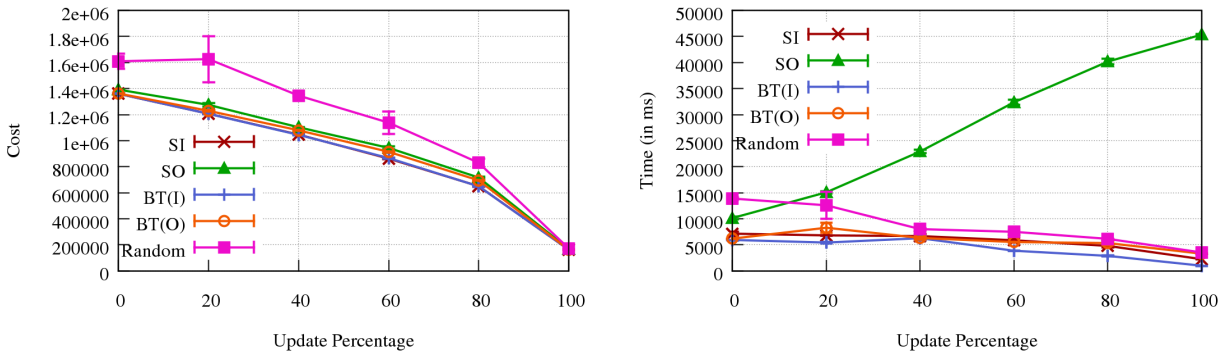
5.1. Setup

Dataset. We generated the dataset from an industry benchmark called YCSB (Yahoo Cloud Servicing Benchmark) [17]. YCSB generates CRUD (create, read, update, delete) operations for benchmarking a key-value store emulating a real-life workload. There are parameters important in YCSB which we explain next. YCSB works in two distinct phases: 1) load: inserts keys to an empty database. The *recordcount* parameter controls the number of inserted keys. 2) run: generates CRUD operations on the loaded database. The *operationcount* parameter controls the number of operations.

Our experiments only consider insert and update operations in order to load memtables (and thus, sstables). We do not consider deletes or reads. Deletes are handled by appending a *tombstone* to the memtable. When compaction encounters a tombstone, it removes the associated key from the database. This handling is very similar to normal update operations and thus, we chose to ignore deletes for simplicity. Reads do not modify sstables and thus, are not generated.

In YCSB, update operations access keys using one of the three realistic distributions: 1) Uniform: All the inserted keys are uniformly accessed, 2) Zipfian: Some keys are more popular than others (power-law), and 3) Latest: Recently inserted keys are more popular (power-law).

Cluster. We ran our experiments in the Illinois Cloud Computing Testbed [5] which is part of the Open Cirrus project [14]. We used a single machine with 2 quad core CPUs, 16 GB of physical memory and 2 TB of disk capacity. The operating system running is CentOS 5.9.



(a) (b)

Figure 7: **Cost and time for compacting sstables generated by varying update percentage with latest distribution.**

Simulator. Our simulator works in two distinct phases. In the first phase, we create sstables. YCSB's load and run phases generates operations which are first inserted into a fixed size (number of keys) memtable. When the memtable is full, it is flushed as an sstable and a new empty memtable is created for subsequent writes. As a memtable may contain duplicate keys, sstables may be smaller and vary in size.

In the second phase, we merge the generated sstables using some of the compaction strategies proposed in Section 4. By default, the number of sstables we merge in an iteration, k is set to 2. We measure the cost and time at the end of compaction for comparison. The cost represents $cost_{actual}$ defined in Section 2. The running time measures both the strategy overhead and the actual merge time.

We evaluate the following 5 compaction strategies:

1) SMALLESTINPUT (SI): In this strategy, we choose the k smallest cardinality sstables at each iteration (Section 4.3.2). We implement this via a priority queue to find the smallest cardinality sstables. This implementation works in $O(\log n)$ time per iteration.
2) SMALLESTOUTPUT (SO): In this strategy, we choose the k sstables whose union has the smallest cardinality (Section 4.3.3). Calculating the cardinality of an output sstable without actually merging the input sstables is tricky. We estimate cardinality of the output sstable using Hyperloglog [18] (HLL). We compute the HLL estimate for all $\binom{n}{k}$ combinations of sstables in the first iteration. At the end of the iteration, k sstables are removed and a new sstable is added. In the next iteration, we have to compute the estimates for $\binom{n-k+1}{k}$ combinations. We can reduce this number by making the following two observations: 1) some of the estimates from the last iteration (involving sstables not removed) can be reused and 2) compute estimate for only those combinations which involve the new sstable. Thus, the total number of combinations for which we need to estimate cardinality is $\binom{n-k}{k-1}$. The per-iteration overhead for this strategy is high.
3) BALANCETREE with SMALLESTINPUT at each level ($BT(I)$): This strategy merges sstables in a single level together (Section 4.3.1). Since all sstables at a single level can be simultaneously merged, we use threads to parallelly initiate multiple merge operations. BALANCETREE does not specify a order for choosing sstables to merge in a single level. We use SMALLESTINPUT strategy and pick sstables in the increasing order of their cardinality.
4) BALANCETREE with SMALLESTOUTPUT at each level ($BT(O)$): This is similar to $BT(I)$ strategy except we use SMALLESTOUTPUT for finding sstables to merge together at each level. Even though, the SO strategy has a large per-iteration strategy overhead, the overhead for this strategy is amortized over multiple iterations that happen in a single level.
5) RANDOM: As a strawman to compare against, we implemented a random strategy that picks random k sstables to merge (at each iteration). This represents the case when there is no compaction strategy. It will thus provide a baseline to compare with.

5.2. Strategy Comparison

We compare the compaction heuristics from Section 4 using real-life (YCSB) workloads. We fixed the operationcount at 100K, recordcount at 1000 and memtable size at 1000. We varied the workload along a spectrum from insert heavy (insert proportion 100% and update proportion 0%) to update heavy (update proportion 100% and insert proportion 0%). We ran experiments with all 3 key access distributions in YCSB.

With 0% updates, the workload only comprises of new keys. With 100% updates, all the keys inserted in the load phase will be repeatedly updated implying a larger intersection among sstables. When keys are generated with a power-law distribution (zipfian or latest) the intersections increase as there will be a few popular keys updated frequently. We present result for latest distribution only. The observations are similar for zipfian and uniform and thus, excluded.

Figures 7 plots the average and the standard deviation for cost and time for latest distribution from 3 independent runs of the experiment. We observe that SI and $BT(I)$ have a compaction cost that is marginally lower than $BT(O)$

(for latest distribution) and SO. Compaction using $BT(I)$ finishes faster compared to SI because of its parallel implementation. RANDOM is the worst strategy. Thus, $BT(I)$ is the best choice to implement in practice. As updates increase, the cost of compaction decreases for all strategies. With a fixed operationcount, larger intersection among sstables implies fewer unique keys, which in turn implies fewer disk writes.

RANDOM is much worse than our heuristics at small update percentage. This can be attributed to the balanced nature of the merge trees. Since sstables are flushed to disk when the memtable reaches a size threshold, the sizes of the actual sstable have a small deviation. Merging two sstables (S_1 and S_2) of similar size with small intersection (small update percentage) creates another sstable (S_3) of roughly double the size at the next level. Both SI and SO choose S_3 for merge only after all the sstables in the previous level have been merged. Thus, their merged trees are balanced and their costs are similar. On the contrary, RANDOM might select S_3 earlier and thus, have a higher cost.

As the intersections among sstables increase (with increasing update percentage), the size of sstables in the next level (after a merge) is close to the previous level. At this point, it is immaterial which sstables are chosen at each iteration. Irrespective of the merge tree, the cost of a merge is constant [2]. Thus, RANDOM performs as well as the other strategies when the update percentage is high.

The cost of SO and $BT(O)$ is sensitive to the error in cardinality estimation. The generated merge schedule differs from the one generated by the naive sstable merging scheme which accurately identifies the smallest union. This results in slightly higher overall cost. The running time of SO increases linearly as updates increase because cardinality estimation overhead increases as intersections among sstables increase.

5.3. Comparison with Optimal

In this experiment, we wish to evaluate how close $BT(I)$, our best strategy, is to optimal. Extensively searching all permutations of merge schedules for finding the optimal cost for large number and size of sstable is prohibitive and exponentially expensive. Instead, we calculate the sum of sstable sizes, our known lower bound for optimal cost from Section 4.1. We vary the memtable size from 10 to 10K and fix the number of sstables to 100. The recordcount for load stage is 1000 and update insert ratio is set to 60:40. The number of operations (operationcount) for YCSB is calculated as: memtable size(10 to 10K) $\times$ number of sstables (100) $-$ recordcount (1000). We ran experiments for all three key access distributions.

Figure 8 compares the cost of merge using $BT(I)$ with the lower-bounded optimal cost, averaged over 3 independent runs of the experiment. Both x and y-axis use log scale. As the maximal memtable size (before flush) increases

2. If sstable size is s, number of sstables to merge in an iteration is 2 and the number of sstables is n, then $cost_{actual}$ would be $3 \cdot (n-1) \cdot s$.

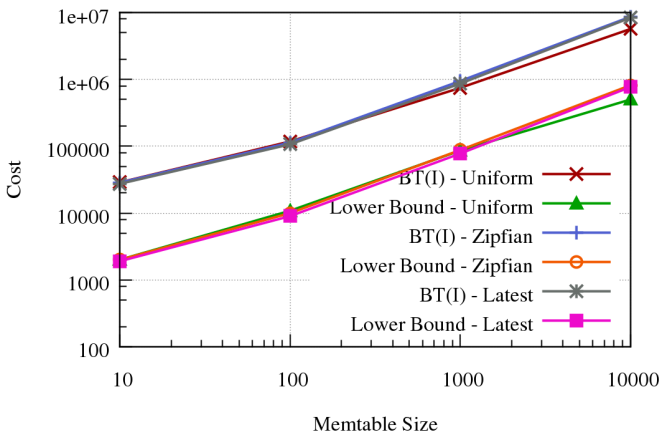


Figure 8: **Comparing cost of $BT(I)$ to optimal which is lower bounded by sum of sizes of all sstables. Both x and y-axis are in log scale.**

exponentially, both the curves show a linear increase in log scale with similar slope. Thus, in real life workloads, the cost of our strategy is within a constant factor of the lower bound of the optimal cost. This is a better performance than the analyzed worst case $O(\log n)$ bound (Lemma 4.5).

5.4. Cost Function Effectiveness

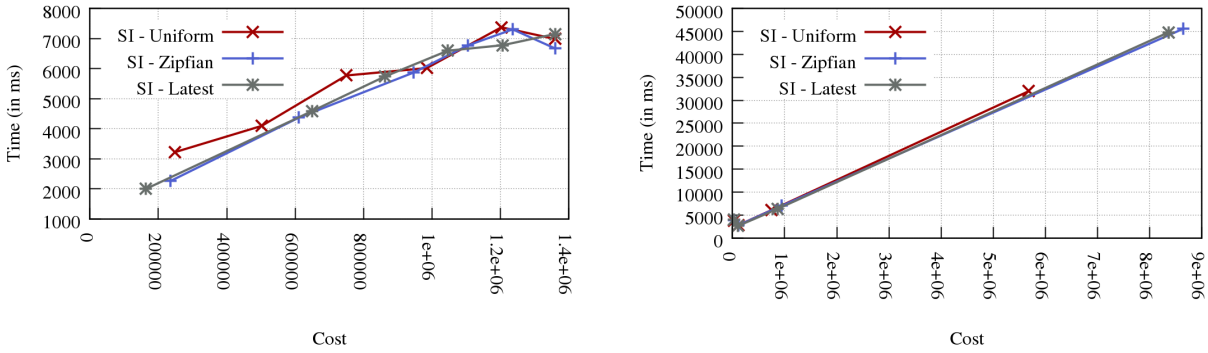
In Section 2 we defined $cost_{actual}$ to model amount of data to be read from and written to disk. This cost also determines the running time for compaction. The goal of this experiment is to validate how the defined cost function affects the compaction time. In this experiment, we compare the cost and time for SI. We chose this strategy because of its low overhead and single-threaded implementation. We ran our experiments with the same settings as described in Section 5.2 and Section 5.3. Cost and time values are calculated by averaging the observed values of 3 independent runs of the experiment.

Figure 9 plots the cost on x-axis and time on y-axis. As we increase update proportion ((Figure 9b) and operationcount (Figure 9a), we see an almost linear increase for time as cost increases for all 3 distributions. This validates the cost function in our problem formulation, as by minimizing it, we will reduce the running time as well.

6. Conclusion

In this work, we formulated compaction in key value stores as an optimization problem. We proved it to be NP-hard. We proposed 3 heuristics and showed them to be $O(\log n)$ approximations. We implemented and evaluated the proposed heuristics using real-life workloads. We found that a balanced tree based approach $BT(I)$ provides the best tradeoff in terms of cost and time.

Many interesting theoretical questions still remain. The $O(\log n)$ approximation bound shown for the



(a) Increasing update percentage

(b) Increasing operationcount

Figure 9: **Effect of cost function on completion time for compaction.** SI **strategy used. Update Percentage and datasize varied for the plots respectively.**

SMALLESTINPUT and SMALLESTOUTPUT heuristic seems quite pessimistic. Under real-life workloads, the algorithms perform far better than $O(\log n)$. We do not know of any bad example for these two heuristics showing that the $O(\log n)$ bound is tight. This naturally motivates the question, if the right approximation bound is infact $O(1)$. Finally, it will be interesting to study the hardness of approximation for the BINARYMERGING problem.

Acknowledgement. We thank Chandra Chekuri for helpful discussions and insightful ideas regarding proofs and plots.

References

[1] Cassandra. http://cassandra.apache.org/. visited on 2014-04-29.

[2] CouchDB. http://couchdb.apache.org. visited on 2014-04-29.

[3] Date Tiered Compaction. https://issues.apache.org/jira/browse/CASSANDRA-6602. visited on 2014-11-20.

[4] HBase. https://hbase.apache.org. visited on 2014-04-29.

[5] Illinois Cloud Computing Testbed. http://cloud.cs.illinois.edu/. visited on 2014-11-20.

[6] Improving compaction in Cassandra with cardinality estimation. http://www.datastax.com/dev/blog/improving-compaction-in-cassandra-with-cardinality-estimation. visited on 2014-12-09.

[7] LevelDB. http://google-opensource.blogspot.com/2011/07/leveldb-fast-persistent-key-value-store.html. visited on 2014-11-24.

[8] LevelDB in Riak. http://docs.basho.com/riak/latest/ops/advanced/backends/leveldb/. visited on 2014-11-24.

[9] Leveled Compaction in Cassandra. http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra. visited on 2014-11-24.

[10] Riak. http://basho.com/riak/. visited on 2014-04-29.

[11] RocksDB. http://rocksdb.org/. visited on 2014-11-20.

[12] Size Tiered Compaction. http://shrikantbang.wordpress.com/2014/04/22/size-tiered-compaction-strategy-in-apache-cassandra/. visited on 2014-11-20.

[13] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[14] R. Campbell, I. Gupta, M. Heath, S. Y. Ko, M. Kozuch, M. Kunze, T. Kwan, K. Lai, H. Y. Lee, M. Lyons, D. Milojicic, D. O'Hallaron, and Y. C. Soh. Open Cirrus Cloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[15] 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. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[16] C. Chekuri and R. Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics*, 98(12):29 – 38, 1999.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[18] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *IN AOFA 07: Proceedings of the 2007 International Conference on Analysis of Algorithms*, 2007.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms (TALG)*, 8(1):4:1–4:22, Jan. 2012.

[20] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.

[21] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[22] L. Lovsz. Submodular functions and convexity. In A. Bachem, B. Korte, and M. Grtschel, editors, *Mathematical Programming The State of the Art*, pages 235–257. Springer Berlin Heidelberg, 1983.

[23] M. J. Luczak and S. Noble. Optimal arrangement of data in a tree dictionary. *Discrete Applied Mathematics*, 113:243–253, 2001.

[24] C. Mathieu, C. Staelin, and N. E. Young. K-slot sstable stack compaction. *CoRR*, abs/1407.3008, 2014.

[25] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: A scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, June 2012.

Appendix A.

We now formally prove that BINARYMERGING is NP-hard.

A.1. The SIMPLE DATA ARRANGEMENT problem

The following problem, known as the SIMPLE DATA ARRANGEMENT problem is discussed in the paper of Luczak and Noble [23]. The problem is defined as follows:
Instance: Given graph $G = (V, E)$ and a nonnegative integer B given in binary.
Question: Is there an injective mapping f from V to the leaves of a complete d-ary tree T, of height $\lceil \log_d |V| \rceil$, such that $\sum_{\{i,j\} \in E} d_T(f(i), f(j)) \leq B$?

Luczak and Noble show in [23], that the above problem is NP-hard for any $d \geq 2$. We notice that in the above, we may assume that $|V|$ is an exact power of d, i.e., $|V| = d^{\log_d |V|}$. The SIMPLE DATA ARRANGEMENT problem reduces to such special cases, and so this variant is also NP-hard.

A.2. A problem related to BINARYMERGING

We consider the following problem which is related to the BINARYMERGING problem. In this problem we are given n sets $A_1, \ldots, A_n$, and a full binary tree T with n leaves, and the problem is to find a labeling function π which assigns the sets to the leaves such that $\mathsf{cost}(T, \pi, A_1, \ldots, A_n)$ is minimized. Notice that the only difference with the BINARYMERGING problem is that here we are fixing the tree T while the BINARYMERGING problem seems somewhat harder. Let OPT-TREE-ASSIGN denote this problem, and for an instance with sets $A_1, \ldots, A_n$ and the full binary tree T with n leaves, let $\mathsf{opt}_a(T, A_1, \ldots, A_n)$ denote the value of optimal solution. Let $n = 2^h$ be a power of 2, and let $\mathcal{T}$ be a perfectly balanced tree with n leaves, and height $h = \log n$. We first show the following

Lemma A.1. *The* OPT-TREE-ASSIGN *problem is NP-hard, for instances where the number of sets n is a power of two, and the tree input T is the balanced binary tree $\mathcal{T}$.*

Proof. Consider a given instance of the SIMPLE DATA ARRANGEMENT problem: we have a graph $G = (V, E)$, where $|V| = n = 2^h$ for some integer h, and an integer B in binary. We define n sets, A_i, one for each vertex i, where we assume that the vertices have been labeled as $1, \ldots, n$. Suppose that $e_1, \ldots, e_k$ are the edges incident to a vertex i. Then we define $A_i = \{e_1, \ldots, e_k\}$. We consider the instance of the OPT-TREE-ASSIGN problem given by the sets A_i for $1 \leq i \leq n$ and the complete binary tree $\mathcal{T}$ with $n = 2^h$ leaf nodes. Suppose $\pi : [n] \to [n]$ is any labeling function, that assigns the set A_i to the leaf numbered $\pi(i)$. Clearly, the size of this instance is only polynomially larger than that of the given instance of the SIMPLE DATA ARRANGEMENT problem, which we assume has size at least $|V| + |E|$. Now we use the reformulation of the cost function for the OPT-TREE-ASSIGN problem as given in Section 2. Suppose $e = (i, j)$ is an edge of G. Clearly e occurs in exactly two of the sets A_i, A_j. Further the cost $\mathcal{T}(e)$ can be seen to be exactly, $\log n + \frac{1}{2} d_{\mathcal{T}}(\pi(i), \pi(j))$. Therefore, the total cost for the OPT-TREE-ASSIGN using the labeling π is,

$$\mathsf{cost}(\mathcal{T}, \pi, A_1, \ldots, A_n) = |E| \log n + \frac{1}{2} \sum_{e=(i,j)} d_{\mathcal{T}}(\pi(i), \pi(j)).$$

Therefore, we have that,

$$\min_{\text{labelings } \pi} \sum_{e=(i,j)} d_{\mathcal{T}}(\pi(i), \pi(j)) = 2\mathsf{opt}_a(\mathcal{T}, A_1, \ldots, A_n) - 2\,|E| \log n$$

Notice that $\min_{\text{labelings } \pi} \sum_{e=(i,j)} d_{\mathcal{T}}(\pi(i), \pi(j))$ is precisely the cost of the best injective mapping for the SIMPLE DATA ARRANGEMENT problem. Therefore, a polynomial time algorithm for the OPT-TREE-ASSIGN problem would help us evaluate this number and then we can decide if it is less than B. Our reduction is complete. ■

A.3. Forcing a complete binary tree

Let T be a binary tree with $n = 2^h$ leaves for some integer h. Let r be the root node of T. Let $\eta(T)$ be the sum of the root to leaf node distances for all the leaf nodes, i.e.,

$$\eta(T) = \sum_{\nu \text{ leaf node of } T} d_T(r, \nu),$$

Lemma A.2. *For any binary tree T with n leaf nodes, we have that $\eta(T) \geq n \log n$ with equality only for the perfect binary tree; here $n = 2^h$ for some integer h.*

Proof. This holds if T has only 1 leaf node. We may assume that T is a full binary tree, otherwise the value of $\eta(T)$ can be decreased. Any full tree must have at least 2 leaf nodes and we can easily verify that the given result holds here. We use induction to prove the result for $n > 2$. Suppose that the root r has two full trees T_1 and T_2 as children with n_1, n_2 leaf nodes respectively. We have, $n = n_1 + n_2$ and, $\eta(T) = \eta(T_1) + \eta(T_2) + n_1 + n_2$. As $n_1, n_2 < n$, we use the induction assumption to get that $\eta(T_1) \geq n_1 \log n_1$ and similarly for T_2. Now using the strict convexity of the function $f(x) = x \log x$ for $x > 0$ we immediately get that $\eta(T) \geq n \log n$. For equality, both T_1 and T_2 must achieve equality and moreover they must have the same number of leaf nodes, i.e., $n_1 = n_2 = 2^k$ for some k. Moreover, they must both be complete trees. This means T is also a complete binary tree with $n = 2^{k+1}$ leaf nodes. ■

Given n sets $A_1, \ldots, A_n$ with m elements, let T be any full binary tree on n leaves, and π be the leaf assignment function. Then the cost of the merge according to T and π, i.e., $\mathsf{cost}(T, \pi, A_1, \ldots, A_n)$ is at most mn^2. To see this notice that any full binary tree has height at most $n-1$. Each element x of the m elements, occurs in at most n sets, and its total cost according to T, i.e., $T(x)$ can be at most n^2. We encapsulate it as the following elementary result,

Lemma A.3. *Let $A_1, \ldots, A_n$ be n sets with a total of m elements. Then, for any full binary tree T,and any labeling π, we have that, $\mathsf{cost}(T, \pi, A_1, \ldots, A_n) \leq mn^2$.*

Let $B_1, \ldots, B_n$ be sets. Consider the instance of the OPT-TREE-ASSIGN problem on the sets $A_1 \cup B_1, \ldots, A_n \cup B_n$, and full binary tree T. We have the following result which is easy to derive using the definitions.

Lemma A.4. *Suppose that the B_i are disjoint from each other and all the A_j, i.e., $B_i \cap B_j = \emptyset$ for $i \neq j$ and $B_i \cap A_j = \emptyset$ for all i, j. Moreover, suppose that $|B_1| = |B_2| = \ldots = |B_n| = S$. Then, for any full binary tree T with n leaves, and for any labeling function $\pi : [n] \to [n]$, we have that,*

$$\mathsf{cost}(T, \pi, A_1 \cup B_1, \ldots, A_n \cup B_n) = \mathsf{cost}(T, \pi, A_1, \ldots, A_n) + S\eta(T).$$

We can now show how we can force a complete binary tree for the BINARYMERGING problem. The result is encapsulated in the following lemma.

Lemma A.5. *Let $A_1, \ldots, A_n$ specify an instance of the* BINARYMERGING *problem where $|A_1 \cup \ldots \cup A_n| = m$, and $n = 2^h$ for some integer h. Suppose that $B_1, \ldots, B_n$ are disjoint sets and disjoint from each of the A_i. Suppose, $|B_1| = \ldots = |B_n| = S$ where $S > mn^2$. Then, in the solution to the* BINARYMERGING *problem for the sets, $A_1 \cup B_1, \ldots, A_n \cup B_n$, the optimal solution must use a complete binary tree. Moreover, we have that,*

$$\mathsf{opt}_a(\mathcal{T}, A_1, \ldots, A_n) = \mathsf{opt}_s(A_1 \cup B_1, \ldots, A_n \cup B_n) - Sn \log n.$$

Proof. Suppose that T is an optimal tree for a solution to the BINARYMERGING problem for the sets $A_1 \cup B_1, \ldots, A_n \cup B_n$, and π is the labeling used in the optimal solution. By Lemma A.4, we have that $\mathsf{cost}(T, \pi, A_1 \cup B_1, \ldots, A_n \cup B_n) = \mathsf{cost}(T, \pi, A_1, \ldots, A_n) + S\eta(T)$. Let $\mathcal{T}$ be a balanced tree and we use the same labeling π. We have, $\mathsf{cost}(\mathcal{T}, \pi, A_1 \cup B_1, \ldots, A_n \cup B_n) = \mathsf{cost}(\mathcal{T}, \pi, A_1, \ldots, A_n) + S\eta(\mathcal{T})$. If T is not the complete tree, we must have by Lemma A.2, we have that $\eta(T) \geq \eta(\mathcal{T}) + 1$. As such

$$\begin{aligned}\mathsf{cost}(T, \pi, A_1 \cup B_1, \ldots, &A_n \cup B_n) \\ &\geq \mathsf{cost}(\mathcal{T}, \pi, A_1 \cup B_1, \ldots, A_n \cup B_n) \\ &\quad + (S - \mathsf{cost}(\mathcal{T}, \pi, A_1, \ldots, A_n)) \\ &\quad + \mathsf{cost}(T, \pi, A_1, \ldots, A_n).\end{aligned}$$

Now, Lemma A.3, and the fact that $S > mn^2$, implies that, $\mathsf{cost}(T, \pi, A_1 \cup B_1, \ldots, A_n \cup B_n) > \mathsf{cost}(\mathcal{T}, \pi, A_1 \cup B_1, \ldots, A_n \cup B_n)$. This contradicts our assumption that T and π realize the optimal solution. As such, T must be the balanced tree $\mathcal{T}$.

Now, using Lemma A.4, we can write for any labeling π, $\mathsf{cost}(\mathcal{T}, \pi, A_1, \ldots, A_n) = \mathsf{cost}(\mathcal{T}, \pi, A_1 \cup B_1, \ldots, A_n \cup B_n) - Sn \log n$. By minimizing both sides over all possible labelings π, we get that, $\mathsf{opt}_a(\mathcal{T}, A_1, \ldots, A_n) = \mathsf{opt}_a(\mathcal{T}, A_1 \cup B_1, \ldots, A_n \cup B_n) - Sn \log n$. However, since the optimal solution for the BINARYMERGING problem for the sets $A_1 \cup B_1, \ldots, A_n \cup B_n$, must use the tree $\mathcal{T}$ as already shown, we have that, $\mathsf{opt}_s(A_1 \cup B_1, \ldots, A_n \cup B_n) = \mathsf{opt}_a(\mathcal{T}, A_1 \cup B_1, \ldots, A_n \cup B_n)$. The equation now follows. ■

The reduction. Our next lemma reduces (special instances of) the OPT-TREE-ASSIGN problem, which is known to be NP-hard, see Lemma A.1, to BINARYMERGING.

Lemma A.6. *The* OPT-TREE-ASSIGN *problem where the number of sets n is a power of two, and the input tree T is the perfectly balanced binary tree $\mathcal{T}$, is polynomial time reducible to the* BINARYMERGING *problem.*

Proof. Consider an instance of the OPT-TREE-ASSIGN problem; we are given n sets $A_1, \ldots, A_n$, where $n = 2^h$ for some integer h. The input tree is $\mathcal{T}$, the perfectly balanced binary tree with n leaf nodes. Let there be m elements in all. The input to the problem is of size at least $\max(n, m)$ since each element requires at least 1 bit and so does each set. We create sets $B_1, \ldots, B_n$ each of which are disjoint from any A_i and they are all disjoint from each other. Moreover, they all have $S = mn^2 + 1$ elements. The sets $A_1 \cup B_1, \ldots, A_n \cup B_n$ now make up an instance of the BINARYMERGING problem. Clearly, we can do this reduction in time polynomial in the input size of the given OPT-TREE-ASSIGN problem. By Lemma A.5, we have that, $\mathsf{opt}_a(\mathcal{T}, A_1, \ldots, A_n) = \mathsf{opt}_s(A_1 \cup B_1, \ldots, A_n \cup B_n) - Sn \log n$. This concludes the reduction. ■

From the reduction in Lemma A.6 the following follows:

Theorem A.7. *The* BINARYMERGING *problem is NP-hard.*

Appendix B.

Lemma B.1. *Given n non-negative integers $x_1, \ldots, x_n$, the sum of the smallest two integers is at most $\frac{2}{n} \cdot \sum_{i=1}^{n} x_i$.*

Proof. Without loss of generality, assume that the numbers are arranged in non-decreasing order, i.e., $x_1 \leq x_2 \leq \ldots \leq x_n$. We have to show $x_1 + x_2 \leq \frac{2}{n} \cdot \sum_{i=1}^{n} x_i$.

Case 1: $n = 2m + 1$ i.e., odd. Add 1 new integer $x_{2m+2} = x_1$. Pair the numbers up from left-to-right, according to its index. The $m + 1$ pairs formed are $(x_1, x_2), \ldots, (x_{2m+1}, x_{2m+2})$. Add numbers in each pair to get $m + 1$ new non-negative integers, $y_1, \ldots, y_{m+1}$. Since x_1 and x_2 were the smallest two numbers, hence $x_1 + x_2$ is the smallest number in the new collection. Then,

$$\begin{aligned}x_1 + x_2 &\leq \frac{1}{m+1} \cdot \sum_{i}^{m+1} y_i = \frac{1}{m+1} \cdot (\sum_{i}^{2m+1} x_i + x_1) \\ &\leq \frac{1}{m+1} \cdot (\sum_{i}^{2m+1} x_i + \frac{1}{m+1} \cdot \sum_{i}^{2m+1} x_i) [\because x_1 \text{ is smallest}] \\ &= \frac{2}{2m+1} \cdot \sum_{i}^{2m+1} x_i = \frac{2}{n} \cdot \sum_{i}^{n} x_i\end{aligned}$$

Case 2: $n = 2m$ i.e., even. This case is much simpler and the argument is similar to the odd case. ■