Cross-Referenced Dictionaries and the Limits of Write Optimization

Peyman Afshani* Michael A. Bender† Martín Farach-Colton‡ Jeremy T. Fineman§

Mayank Goswami¶ Meng-Tsung Tsai‖

Abstract

Dictionaries remain the most well studied class of data structures. A dictionary supports insertions, deletions, queries, and usually successor, predecessor, and extract-min. In a RAM, all such operations take $\mathcal{O}(\log N)$ time on N elements.

Dictionaries are often cross-referenced as follows. Consider a set of tuples $\{\langle a_i, b_i, c_i \ldots \rangle\}$. A database might include more than one dictionary on such a set, for example, one indexed on the a's, another on the b's, and so on. Once again, in a RAM, inserting into a set of L cross-referenced dictionaries takes $\mathcal{O}(L \log N)$ time, as does deleting.

The situation is more interesting in external memory. On a Disk Access Machine (DAM), B-trees achieve $\mathcal{O}(\log_B N)$ I/Os for insertions and deletions on a single dictionary and K-element range queries take optimal $\mathcal{O}(\log_B N + K/B)$ I/Os. These bounds are also achievable by a B-tree on cross-referenced dictionaries, with a slowdown of an L factor on insertion and deletions.

In recent years, both the theory and practice of external-memory dictionaries has been revolutionized by ***write-optimization*** techniques. The best (and optimal) dictionaries achieve a substantially improved insertion and deletion cost of $\mathcal{O}(\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}})$, $0 \leq \varepsilon \leq 1$, amortized I/Os on a single dictionary while maintaining optimal $\mathcal{O}(\log_{1+B^\varepsilon} N + K/B)$-I/O range queries.

Although write optimization still helps for insertions into cross-referenced dictionaries, its value for deletions is greatly reduced. A deletion into a cross-referenced dictionary only specifies a key a. It seems to be necessary to look up the associated values $b, c \ldots$ in order to delete them from the other dictionaries. This takes $\Omega(\log_B N)$ I/Os, well above the per-dictionary write-optimization budget of $\mathcal{O}(\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}})$ I/Os. So the total deletion cost is $\mathcal{O}(\log_B N + L\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}})$ I/Os.

In short, for deletions, write optimization offers an advantage over B-trees in that L multiplies a lower order term, but when $L = 2$, write optimization seems to offer no asymptotic advantage over B-trees. That is, no known solution for pairs of cross-referenced dictionaries seem to beat B-trees for deletions.

In this paper, we show a lower bound establishing that a pair of cross-referenced dictionaries that supports deletions cannot match the write optimization bound available to insert-only dictionaries.

This result thus establishes a limit to the applicability of write-optimization techniques, which are the basis of many new databases and file systems.

*MADALGO, Aarhus University. Email: peyman@madalgo.au.dk.

†Department of Computer Science, Stony Brook University. Email:bender@cs.stonybrook.edu.

‡Department of Computer Science, Rutgers University. Email: farach@cs.rutgers.edu.

§Department of Computer Science, Georgetown University. Email: Jeremy.Fineman@georgetown.edu.

¶Max-Planck Institute for Informatics. Email: gmayank@mpi-inf.mpg.de.

‖Department of Computer Science, Rutgers University. Email: mtsung.tsai@rutgers.edu.

1 Introduction

Dictionaries remain the most well studied class of data structure. A dictionary supports insertions, deletions, queries, and usually successor, predecessor, and extract-min. But surprisingly basic questions about dictionaries remain unanswered.

These questions arose in the pre-computer era whenever peopled indexed large collections of data. The library of Alexandria is thought to have contained over 600,000 volumes, partitioned first into seven broad topics and then shelved alphabetically by author [12, 27]. Each volume is thought to have had tags called ***pinakes***,[1] which contained metadata. Pinakes were also compiled into a separate volume, which is thought to have been the first library catalog. Thus, people could search for books using the pinakes, but only by scanning through the pinakes in $\langle$*subject*, *author*$\rangle$ order. It was many centuries before there were any libraries of size comparable to that of Alexandria after that library was destroyed, and during that time, libraries were indexed using content-addressable-memory systems— that is, "curators, slaves or freedmen" [14, 17].

Circa 1295, the library at the Collège de Sorbonne at the Université de Paris introduced indexes [23], in the sense that there were volumes compiled for the purpose of locating books according to a variety of criteria. For the first time it became possible to search the content of a library according to distinct orders (by author, subject, or collection[2]), while the books themselves were stored on the shelves in any convenient order. In 1791, Enlightenment thinkers of the French Revolution introduced card catalogs as indexes, making it easier for the indexes to track the changing collection [13].

Today books are stored on the shelves according to a subject-classification scheme (usually the Dewey Decimal System [9] or the Library of Congress Classification System (LC) [19]) to allow for browsing, but they are also indexed in other orders (author, title, subject, keyword). Each particular index on the books is a dictionary ordered by a different key.

Specifying the Required Operations of a Dictionary. The actual data structure at work organizing a library is not merely a set of dictionaries, but a ***system*** of ***cross-referenced dictionaries***, which we call a ***compound dictionary***. We call a compound dictionary an ***L-dictionary*** if it consists of L cross-referenced dictionaries. A compound dictionary maintains a ***cross-reference invariant***, where each dictionary—which we sometimes call an ***index***—stores the same set of items but orders them according to a different comparison function. Thus, every time that a book is inserted into or deleted from the library, each index needs to be updated.

An abstraction of a compound dictionary is as follows. The L-dictionary maintains a set $\mathcal{S} \subseteq \mathcal{U}_1 \times \mathcal{U}_2 \times \cdots \times \mathcal{U}_L$. Each (potentially infinite) key space $\mathcal{U}_i$ is totally ordered. Items can be inserted, deleted, and queried:

- INSERT($\boldsymbol{x}$): $\mathcal{S} \longleftarrow \mathcal{S} \cup \{\boldsymbol{x}\}$. That is, add $\boldsymbol{x}$ to $\mathcal{S}$.
- DELETE(i, x): $\mathcal{S} \longleftarrow \mathcal{S} - \mathcal{U}_1 \times \cdots \times \mathcal{U}_{i-1} \times \{x\} \times \mathcal{U}_{i+1} \times \cdots \times \mathcal{U}_L$.
  That is, remove all tuples $\langle *, \ldots, *, x, *, \ldots, * \rangle$ whose ith component is x.
- LOOKUP(i, x): return $\mathcal{S} \cap \mathcal{U}_1 \times \cdots \times \mathcal{U}_{i-1} \times \{x\} \times \mathcal{U}_{i+1} \times \cdots \times \mathcal{U}_L$.
  That is, return all tuples $\langle *, \ldots, *, x, *, \ldots, * \rangle$ whose ith component is x.

[1] "Pinakes" is ancient Greek for "tables", and is thus consistent with modern database nomenclature.
[2] No title indexes were compiled, since titles were not fixed at that time [22].

- RANGE(i, r_1, r_2): return $\mathcal{S} \cap \mathcal{U}_1 \times \cdots \times \mathcal{U}_{i-1} \times [r_1, r_2] \times \mathcal{U}_{i+1} \times \cdots \times \mathcal{U}_L$, where $[r_1, r_2] = \{x \mid r_1 < x < r_2\}$.
  That is, return all tuples $\langle *, \ldots, *, x, *, \ldots, * \rangle$ in $\mathcal{S}$ for which $r_1 < x < r_2$.

We refer to the index on $\mathcal{U}_i$ as $\mathbf{I}_i$.

Observe that compound dictionaries are distinct from multi-dimensional indexes because delete and query operations on compound dictionaries specify only a single coordinate, whereas delete and query operations on a multi-dimensional dictionary might allow all or some of the coordinates of the deleted item or queried rectangle to be specified.

Compound Dictionaries in Databases. The compound dictionary is one of the most (if not the most) widely used data-structural abstractions, because it appears in essentially every relational database management system (RDBMS). In database terminology, indexes are sometimes also called ***tables***, and the elements that are inserted and deleted are typically called ***rows***.

The actual specification of a database is slightly different: indexes can be defined on tuples of fields; deletions can only be specified on so-called ***primary keys***; and in some databases, only U_1 can be primary; some fields may not have any index associated with them; etc. Our version of the problem is similar enough to capture the essential algorithmic challenge of compound dictionaries.

The Complexity of Deletes in a Compound Dictionary. Considering that compound dictionaries have been around for 720 years and are the basic data structure of databases, it may seem surprising that the algorithmic literature is largely silent on this data structure.

On the other hand, at first glance, there's not that much to say. Insertions, for example, into an L-dictionary are simply L times slower than an insertion into a single dictionary, on both a RAM and in external memory.

Now consider deletes. On a RAM, deletions take $\mathcal{O}(\log N)$ on a dictionary and $\mathcal{O}(L \log N)$ on an L-dictionary. As with insertions, simply decompose one deletion into L such operations.

Even in external memory, the problem seemed trivial until recently. The B-tree [2] achieves optimal $\mathcal{O}(\log_B N + K/B)$ I/Os for range queries on K elements, and insertion and deletion cost of $\mathcal{O}(\log_B N)$ on a dictionary. On an L-dictionary, the cost of deletions is $\mathcal{O}(L \log_B N)$. Once again, a deletion to the compound dictionary is a deletion into each dictionary.

But a little bit more is actually going on, because a deletion seems to require a search. Consider a 2-dictionary on $U_1 \times U_2$. An insertion of $\langle u, v \rangle$ consists of adding $\langle u, v \rangle$ to $\mathbf{I}_1$ ordered by u and into $\mathbf{I}_2$ ordered by v. A deletion DELETE$(1, u)$ seems to require LOOKUP$(1, u)$ to fetch the pair $\langle u, v \rangle$, followed by removing $\langle u, v \rangle$ from both $\mathbf{I}_1$ and $\mathbf{I}_2$. In short, an actual delete from a constituent index requires knowing the key to be deleted. But this seems to require a query to find all the necessary keys.

For a B-tree, this is not a problem. We get the desired bounds by noting that deletions take the same amount of time as searches. One query to get all keys does not slow down the L deletions into individual dictionaries.

Compound Dictionaries and Write Optimization. In recent years, both the theory and practice of external-memory dictionaries have been revolutionized by ***write-optimization*** techniques. The best (and optimal) dictionaries achieve a substantially improved insertion and deletion cost of $\mathcal{O}(\frac{\log_{1+B^{\varepsilon}} N}{B^{1-\varepsilon}})$ amortized I/Os, for $0 \leq \varepsilon \leq 1$, while maintaining optimal $\mathcal{O}(\log_{1+B^{\varepsilon}} N + K/B)$ I/Os for range queries [3, 5–7, 20].

Such techniques are now widespread in the database world [1, 8, 11, 18, 24, 25] and are starting to have an impact on file systems [10, 15, 16, 21, 28].

Deletes and Write Optimization. These systems show a marked asymmetry between insertions/deletions and queries. This asymmetry introduces an algorithmic issue with compound dictionaries.

An L-dictionary composed of a write-optimized dictionaries (WODs) will take time $\mathcal{O}(L\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}})$ to insert into all indexes. However, consider the deletion algorithm, which includes a search. Searches are much slower than insertions, and so the time to delete is $\mathcal{O}(\log_{1+B^\varepsilon} N + L\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}})$. Write optimization does help, because the L multiplies a low-order term, but deletions do not enjoy the full benefits of write optimization.

The alternative is to push the slowdown to the query: one could keep a data structure of all the deletions. Suppose that there is a set $D = \{d_1, d_2, ..., d_\ell\}$ of deletion $\text{DELETE}(1, d_i)$. A query $\text{RANGE}(2, x, y)$ considers a sequence $\langle a_i, b_i \rangle$, where $x < b_i < y$. Some of these a_i might belong to D, and any such pair would need to be filtered out of the answer. These lookups in a data structure on D would slow down the queries, thus yielding deletions that match the write-optimization bound but suboptimal queries.

In either case, the crux of the difficulty seems to be the jump from a single dictionary to a 2-dictionary. In the remainder of the paper, we will therefore restrict our attention to 2-dictionaries when talking about compound dictionaries.

Deletes and Databases. So far, we have described the problem of deletes in write-optimized indexes. This problem is of algorithmic interest, certainly, because the run-time of deletes is a big gap in our understanding of indexing. However, we did not come to this problem originally from a consideration of algorithmic issues. Instead, while building TokuDB [26], we had to deal with the issue of deletions. Deletions are a big problem in the design of write-optimized storage systems. What is particularly interesting to us in this problem is that the pragmatics of building a database so exactly line up with the algorithmics of compound dictionaries.

Warming up. Before we consider the problem of deletes in 2-dictionaries, we begin with the simpler ***count*** problem on single dictionaries. In its simplest version, the count of a dictionary returns the cardinality of the set $\mathcal{S}$ being indexed.

In many instantiations of a dictionary, such as in a database, dictionaries support overwrite insertions, in which a new insertion with the same key replaces the old key. (Actually, the value associated with the key replaces the old value). In RAM, such operations takes $\mathcal{O}(\log N)$ time, and counts can be computed in $\mathcal{O}(\log N)$ time. In a B-tree, such operations take $\mathcal{O}(\log_B N)$ I/Os, and counts can be computed in $\mathcal{O}(\log_B N)$ I/Os.

In a WOD, however, insertions take very few I/Os compared to queries. There are not enough I/Os in an insertion to resolve whether a particular insertion is a new insertion or an overwrite. It seems that we need a query to resolve this issue, either at the time of insertion or at query time, in order to achieve an accurate count. Once again, the asymmetry between the cost of insertions and the cost of queries in a WOD seems to cause some algorithmic problems for some operations.

Our results. In this paper, we warm up by showing that the count operation is slow if insertions are write optimized. Specifically, we show that it is impossible to achieve $\mathcal{O}(\log_B N)$ I/Os for count in the external-memory comparison model unless insertions take $\Omega(\log_B N)$. That is, no write optimization is possible at all for this problem. This result serves as both a first proof on the limits of write optimization and as an simplified proof that shows some of the techniques of the main result. The details can be found in Section 2.

In Section 3, we establish limits on write optimization for deletions on 2-dictionaries. We prove that one may either achieve write-optimized deletes or optimal range queries but not both. Our result is not as general as the count result, because our lower bound establishes that some parts of the write-optimization tradeoff curve are not achievable, whereas in the count lower bound, we show that no write optimization at all is achievable. We conjecture that if range queries are optimal, then deletes takes $\Omega(\log_B N)$ (in the I/O comparison model). We leave this conjecture for future work.

2 Counts and Dictionaries

We define ***1-D count problem*** as follows:

- Static Insertion Phase: Preprocess set $\mathcal{S} = \{a_1, a_2, \ldots, a_N\}$.
- Dynamic Insertion Phase: Insert a sequence of $\sqrt{N}$ elements $\mathcal{D} = \{d_1, d_2, \ldots, d_{\sqrt{N}}\}$.
- Counting Phase: Output the count, $|\mathcal{S} \cup \mathcal{D}|$.

Theorem 1. *In the comparison-based external-memory model, for any algorithm that solves the 1-D count problem using $\mathcal{O}(N \log_B N)$ I/Os for the static insertion phase, there is a constant $c <!$ so that if it performs at most $c\sqrt{N}\log_B N$ I/Os for the dynamic insertion phase, it must perform $\Omega(\sqrt{N}\log_B N)$ I/Os to output the count, $|\mathcal{S} \cup \mathcal{D}|$, in the worst case.*

Proof. Let the sorted order of $\mathcal{S}$ be $a_1 < a_2 < \ldots < a_N$. Suppose the adversary reveals that each d_k is in a disjoint subrange of $\mathcal{S}$ as follows: $a_1 \le d_1 \le a_{\sqrt{N}}, a_{\sqrt{N}+1} \le d_2 \le a_{2\sqrt{N}}, \ldots, a_{(\sqrt{N}-1)\sqrt{N}+1} \le d_{\sqrt{N}} \le a_N$. In the comparison-based model, the only information that the algorithm can learn about d_k is the set of possible a_i that might match d_k, i.e. that $d_k = a_i$, for some i, or that there it lies in some open interval (a_i, a_j), but the relative order of a_{i+1} and d_k is unknown, (and symmetrically with a_{j-1}). We say that d_k is ***resolved*** if we know that $d_k = a_i$ or $d_k \in (a_i, a_{i+1})$, for some i. Otherwise it is ***unresolved*** on some interval (a_i, a_j), $j > i+1$. We note here that in this setting, the algorithm knows that $d_1 < d_2 < \cdots < d_{\sqrt{N}}$, so no extra information can be inferred by comparing pairs of elements in $\mathcal{D}$.

Suppose that the adversary reveals to the algorithm the additional information $|\mathcal{S} \cup \mathcal{D}|$ is either $N + \sqrt{N}$ or $N + \sqrt{N} - 1$. That means at most one member of $\mathcal{D}$ matches some member of $\mathcal{S}$.

To distinguish between the two cases, the algorithm needs to identify the predecessors and successors for the each d_k. To see this, suppose some d_k is unresolved interval (a_i, a_j), $j > i+1$, after all the pre-count I/Os are performed. For all resolved members of $\mathcal{D}$, the adversary can choose to have those elements be distinct from those of $\mathcal{S}$. Thus we must be able to determine if d_k belongs to $\mathcal{S}$. At this point, we therefore must know if $d_k = a_{i+1} < a_j$ or not.

Therefore, we do not know if $d_k \in \mathcal{S}$ if d_k is unresolved. But if d_k is resolved, we know the successor and predecessor of d_k. Hence, to report the count, the algorithm needs to identify the predecessors and successors for each d_k. It is known that finding the predecessors for each d_k requires $\Omega(\sqrt{N}\log_B N)$ (say, at least $c^+\sqrt{N}\log_B N$) I/Os [4, Theorem 7]. Since the I/Os spent on the second phase is $c\sqrt{N}\log_B N$, the third phase must pay off the difference $c^+\sqrt{N}\log_B N - c\sqrt{N}\log_B N = \Omega(\sqrt{N}\log_B N)$, hence proving the theorem. $\square$

3 Deletes and 2-Dictionaries

In this section we show that a 2-dictionary supporting optimal range queries cannot achieve the write-optimization bound ($\mathcal{O}(\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}})$) for $\varepsilon > 2/3$. Brodal and Fagerberg [6] established that external memory dictionaries could not surpass the write-optimization tradeoff bounds, but here we show that compound dictionaries they cannot meet the optimal write-only tradeoff bound in general for deletes.

Specifically, we show a lower bound for ***2-Dictionary Deletion Problem*** (2DD), which we define as the problem of satisfying the following set of compound-dictionary commands:

- Phase 1: A set $\mathcal{S} = \{\langle a_i, b_j \rangle\}$ of N insertions $\textsc{insert}(a_i, b_j)$. Define π by $a_i = \pi(b_j)$. Let $\mathcal{A} = \{a_i\}$ and $\mathcal{B} = \{b_i\}$.
- Phase 2: A set $\mathcal{D} = \{d_i\} \subseteq \mathcal{A}$ of $\sqrt{N}$ deletions $\textsc{delete}(1, d_i)$ on the first coordinate.
- Phase 3: A range query $\textsc{range}(2, b_\ell, b_r)$ on the second coordinate.

Note that the insertions and deletions can be performed in any order, rather than in order by a_i and d_i.

Our main result is the following theorem:

Theorem 2. *For any data structure that solves the 2DD problem, if insertion takes amortized $\mathcal{O}(\log_B N)$ I/Os and deletion takes amortized $\mathcal{O}(\log_B N/B^\alpha)$ I/Os for any constant $\alpha > 2/3$, then some range query* $\textsc{range}(2, b_\ell, b_r)$ *requires $\omega(\log_B N + K/B)$ I/Os, where $K = |\mathcal{B} \cap [b_\ell, b_r]|$ is the number of b's inserted in the range $[b_\ell, b_r]$.*

3.1 Proof Outline

As in the proof of Theorem 1, we specify that the members of $\mathcal{D}$ come from disjoint ranges of $\mathcal{A}$, each of size $\sqrt{N}$. We perform the allowed I/Os and find the uncertainty ranges for all the deletions. In this proof, we need to more carefully quantity the uncertainty that remains in all the deletions, because we need this uncertainty to be large enough to increase the I/Os of a range query.

In other words, counts take $O(\log_B N)$ I/Os, whereas range queries take $O(\log_B N + K/B)$. If K is large, then this term dominates the I/O complexity of a range query, and K can be as large as N. Therefore, we need to prove a much higher lower bound to establish our theorem. Specifically, it is not enough to simply figure out that there are unresolved deletions. We need to make sure that the I/Os required to completely resolve the deletion cannot be amortized against those used to answer the range query.

To begin, we need to replace Theorem 7 from [4], which states that not all searches can fully resolve in less than $c \log_B N$ per query, for some constant c, with a more quantitative bound, which quantifies the amount of uncertainty of the deletions at the end of the deletion phase. (See Lemma 4.)

Because the remaining uncertainty is large, there are many tuples in $\mathbf{I}_2$ that might be deleted. We want to find a range query that has many such potential deletions. We show that such ***hot*** regions exist in Lemma 6. We need to show that there exists such a hot region that involves a sufficient number of different d_i's. In addition, not too many of these d_i's can be fetched into memory as a byproduct when the algorithm fetches other d_i's. To guarantee this, we require π to satisfy two conditions, CA and CB, where Lemma 3 guarantees the existence with the probabilistic method and Turán theorem.

3.2 Preliminaries

We assume that every I/O can read/write a disk page capable of storing $B = \log^\tau N$ tuples for some sufficiently large constant τ, the physical memory can store $M = O(N^\mu)$ tuples, and the range query is of size $K = N^\delta$ and $\mu < \delta < 1/4$ are constants.

Specifying the insertions and deletions. Let $a_1, \ldots, a_N$ be the sorted order of $\mathcal{A}$ and $b_1, \ldots, b_N$ the sorted order of $\mathcal{B}$. We assume that $a_i \neq a_{i+1}$ and $b_i \neq b_{i+1}$, for $1 \leq i < N$, that is, $\mathcal{A}$ and $\mathcal{B}$ each have N distinct values. Recall that the N inserted tuples be $(\pi(b_i), b_i)$ for $i \in [N]$, where π is a mapping from $\{b_1, \ldots, b_N\}$ to $\{a_1, \ldots, a_N\}$.

We will break $\mathcal{A}$ into blocks of size $\sqrt{N}$ as we did in the proof of Theorem 1. We say that a_i has ***color*** k, abbreviated as $c(a_i) = k$, if $\lceil i/\sqrt{N} \rceil = k$. We say b_i has color k if $\pi(b_i)$ has color k and overload the color function so that $c(T) = \{c(b_i) : b_i \in T\}$. For every fixed constant $r \in (0, 1)$, we define the sets

$$S_{t,N^r} = \{b_i : \lceil i/N^r \rceil = t\}.$$

Not every π admits a range query that requires superlinear number of I/Os. Consider, for example, the degenerate case that $\pi(b_i) = a_i$ for all $i \in [N]$. If $\pi(b_i)$ is directly computable from a_i with no I/Os, then the theorem does not hold. We can insert a deletion message into both indices and write optimization works just fine.

Hence, to prove the theorem, we cannot choose π arbitrarily. We will pick a π that satisfies the following two conditions.

CA. $c(b_i) \neq c(b_j)$ if $b_i \neq b_j$ and $b_i, b_j \in S_{t,\sqrt{N}}$ for some $t \in [\sqrt{N}]$.

In other words, the permutation must be compatible with the following: Break $\mathcal{B}$ into blocks of size $\sqrt{N}$ in order. Take the elements of each of these blocks and map them to some element in $\mathcal{A}$, so that no two elements in the same block of $\mathcal{B}$ fall within the same block of size $\sqrt{N}$ in $\mathcal{A}$.

CB. $|c(S_{t,N^\delta}) \cap c(S_{t',N^\delta})| = \mathcal{O}(1)$ for every $t \neq t' \in [N^{1-\delta}]$, for some fixed constant $\delta \in (0, 1/4)$.

We note here that $|c(S_{t,N^{1/2}}) \cap c(S_{t,N^{1/2}})| = \sqrt{N}$ given CA. However, if we break $\mathcal{B}$ into finer blocks of size N^δ, then the pairwise intersection can have size as small as a constant, and in fact, we require it to. The following lemma shows the existence of such a π.

Lemma 3. *For every N and $\delta \in (0, 1/4)$, there exists some π that satisfies both* CA *and* CB.

Proof. To satisfy CA, it is required that $c(S_{t,\sqrt{N}}) = \{1, 2, \ldots, \sqrt{N}\}$ for every $t \in [\sqrt{N}]$. Hence, $c(b_1), c(b_2), \ldots, c(b_N)$ is a concatenation of $\sqrt{N}$ permutations of $1, 2, \ldots, \sqrt{N}$.

There are $(\sqrt{N})!$ such permutations but, to satisfy CB, some permutations cannot be placed together in the concatenation. We construct a graph $G = (V, E)$ to describe which permutations cannot be placed together. Each node in G denotes a permutation and thus $|V| = (\sqrt{N})!$. If two permutations cannot be placed together in the concatenation due to CB, then we connect the representative nodes by an edge. Because of symmetry, the graph is regular.

Here we upper bound the degree of each node. Let π_0 be a permutation specified by some fixed node and π_{rand} be a permutation specified by the node picked uniformly at random. Let T be the threshold constant in CB (i.e. $|c(S_{t,N^\delta}) \cap c(S_{t',N^\delta})| < T$). Then, each node in G has degree

$$\begin{aligned}
d &= \left(\sqrt{N}\right)! \cdot \Pr[\pi_0 \text{ and } \pi_{rand} \text{ cannot be placed together}] \\
&\le \left(\sqrt{N}\right)! \sum_{\substack{i_1,i_2,\ldots,i_T \in [\sqrt{N}] \\ i_1 \neq i_2 \neq \cdots \neq i_T}} \Pr[i_1, i_2, \ldots, i_T \text{ fall in the same chunk of size } N^\delta \text{ in } \pi_0 \text{ and } \pi_{rand}] \\
&= \left(\sqrt{N}\right)! \left(N^{1/2-\delta}\right) \binom{N^\delta}{T} \left(\frac{1}{N^{1/2-\delta}}\right)^T \\
&\le \left(\sqrt{N}\right)! \left(N^{1/2-\delta-T(1/2-2\delta)}\right) && \text{(note that } \delta < 1/4\text{)} \\
&\le \left(\sqrt{N}\right)!/N && \text{(pick a sufficiently large } T\text{)}
\end{aligned}$$

By Turán Theorem, the graph $G = (V, E)$ has an independent set of size at least

$$\frac{|V|^2}{|V| + 2|E|} \ge \frac{\left(\left(\sqrt{N}\right)!\right)^2}{\left(\sqrt{N}\right)! + \left(\left(\sqrt{N}\right)!\right)^2/N} = N - o(1),$$

meaning that some carefully chosen $\sqrt{N}$ permutations can be placed together in the concatenation without violating CB. As a result, the desired π exists. $\square$

Given a mapping π that satisfies the both conditions, the adversary conducts the following ***adversarial sequence of insertions and deletions***:

- Insertion Phase: The adversary inserts, in any order, N tuples $(\pi(b_i), b_i)$ for every $i \in [N]$.
- Deletion Phase: The adversary deletes, in any order, $\sqrt{N}$ tuples $(d_k, *)$ for every $k \in [\sqrt{N}]$, where $d_k = a_i$ for some a_i whose color is k.

At the beginning of the deletion phase, each d_k, for $k \in [\sqrt{N}]$, might match any a_i whose color is k. We say that d_k has ***uncertainty*** u, abbreviated as $U(d_k) = u$, if the number of a_i's that can match d_k equals u. While performing the I/Os for deletions, some comparisons between a's and d's are made and thus the uncertainly $U(d_k)$ of any d_k might shrink, but we claim that not by too much, in aggregate, of all k. Here we prove a quantitative bound for the sum of the $U(d_k)$ at the end of the deletion phase.

Lemma 4. *Any algorithm for 2DD over an adversarial sequence using amortized $\mathcal{O}(\log_B N)$ I/Os per insertion and $\mathcal{O}(\log_B N/B^\alpha)$ I/Os per deletion, for any constant $\alpha > 2/3$, has*

$$\sum_{k \in [\sqrt{N}], U(d_k) > 1} U(d_k) = \Omega(N/B^{1-\alpha})$$

at the end of the deletion phase.

Proof. It suffices to show that the desired lower bound holds even if the adversary reveals some information for each d_k at the beginning of the deletion phase. For each $k \in [\sqrt{N}]$, the adversary partitions the range $[(k-1)\sqrt{N} + 1, k\sqrt{N}]$ into B^r equal-sized consecutive subranges for some constant r determined later. It then randomly picks a subrange and reveals to the algorithm that d_k equals some a_i in the subrange. After such a revelation,

$$\sum_{k \in [\sqrt{N}], U(d_k) > 1} U(d_k) = \Omega(N/B^r).$$

Claim 5. *For some combination of randomly picked subranges and $r < 1/3$, the sum of uncertainty $\Omega(N/B^r)$ cannot be further narrowed down by any superconstant factor at the end of the deletion phase.*

Proof. Let us consider the I/Os performed during the deletion phase. These I/Os can bring a's from disk to memory for subsequent comparisons with d's, and thus the uncertainty of d's can be reduced. For each a that is brought into memory, it is only possible to reduce the uncertainty of one d. We assume that the adversary reduces the uncertainty of the appropriate d, even if that d isn't in memory. Thus, we give the algorithm more power than any actual algorithm could have.

We say that some a is ***fresh*** if it has not yet been brought into memory since the beginning of the deletion phase, and therefore only fresh a's can be used to reduce the uncertainty further given the assumption. We note that only fetching the disk pages written in the insertion phase can give the algorithm fresh a's. Those written in the deletion phase cannot, since all uncertainty is maximally reduced when an a is fetched during the deletion phase. There are $\mathcal{O}(N \log_B N)$ disk pages written in the insertion phase, and therefore the number of disk pages that can contain some fresh a's is $\mathcal{O}(N \log_B N)$.

Let $P_i \subseteq \{a_j : j \in [N]\}$ denote the a_j's contained in the ith disk page written in the insertion phase. Note that $|P_i| \leq B$. Let $P_{i,k} = \{a_j \in P_i : c(a_j) = k\}$. We partition P_i into two disjoint sets H_i and L_i, where $L_i = P_i \setminus H_i$ and

$$H_i = \{a_j \in P_i : |P_{i,c(a_j)}| \geq B^r\}.$$

That is H_i is the set of elements in P_i whose color is frequently represented in P_i. Let R_k be the randomly picked subrange for d_k. Let $X_{i,k}$ denote the random variable $|R_k \cap P_i|$. Then $X_{i,k} \in [0, |P_{i,k}|]$, $\mathbb{E}[X_{i,k}] = |P_{i,k}|/B^r$, and all $X_{i,k}$'s are independent for every fixed i. Let

$$Y_i = \sum_{k \in [\sqrt{N}], P_{i,k} \subseteq L_i} X_{i,k},$$

and from linearity of expectation

$$\mathbb{E}[Y_i] = \sum_{k \in [\sqrt{N}], P_{i,k} \subseteq L_i} \mathbb{E}[X_{i,k}] = \frac{|L_i|}{B^r}.$$

By Hoeffding's inequality, we have

$$\Pr\left[Y_i - \mathbb{E}[Y_i] \geq B^{1-r}\right] \leq \exp\left(-\frac{2(B^{1-r})^2}{\sum_{k \in [\sqrt{N}], P_{i,k} \subseteq L_i} |P_{i,k}|^2}\right) \leq e^{-\Omega\left(\frac{B^{2-2r}}{B^{1+r}}\right)},$$

which is $e^{-B^{\Omega(1)}} = e^{-\log^{\Omega(\tau)} N} = 1/N^2$ if we pick any constant $r < 1/3$ and pick τ to be sufficiently large. By the union bound, we know that for some combination of R_k for $k \in [\sqrt{N}]$,

$$Y_i \leq \mathbb{E}[Y_i] + B^{1-r} \leq 2B^{1-r} \text{ for every disk page written in the insertion phase.}$$

The adversary picks some such combination of R_k, and reveals the information to the algorithm. No matter what $\mathcal{O}(\sqrt{N} \log_B N/B^\alpha)$ I/Os are fetched by the algorithm in the deletion phase — w.l.o.g. let them be $P_1, P_2, \ldots, P_T$ for $T = \mathcal{O}(\sqrt{N} \log_B N/B^\alpha)$ — we have:

- The number of colors contributed by H_i for $i \in [T]$ is at most $(\sqrt{N} \log_B N/B^\alpha)(B/B^r)$.
- The number of a_j's contributed by L_i for $i \in [T]$ is at most $(\sqrt{N} \log_B N/B^\alpha)(2B^{1-r})$.

- The number of a_j's is in memory at the beginning of the deletion phase is at most $M = o(N^\delta)$.

If we pick $\alpha > 1 - r$, there are $o(\sqrt{N})$ d_k's whose uncertainty can be further narrowed down by the a_j's fetched by some H_i. Furthermore, the number of a_j's contained in some L_i and in memory at the beginning of the deletion phase is bounded by $o(\sqrt{N})$, which means that few d_k's can have a comparison with a_j in some L_i to further narrow down the uncertainty. Since r can be any constant $< 1/3$, then α can be any constant $> 2/3$. □

By Claim 5, we complete the proof of Lemma 4. □

After performing all deletions, the number of disk pages that contain some d_i for $i \in [\sqrt{N}]$ is at most $(\sqrt{N}\log_B N)/B^\alpha$ (i.e. no more than the budget of I/Os for deletions).

We are now in a position to prove the existence of a range query that requires superlinear number of I/Os. Observe that if a range query contains some b_j whose $\pi(b_j)$ still might match some d_i, to answer the range query correctly, the algorithm must, due to CA: (1) fetch some disk page that contains d_i, and (2) compare b_j with d_i to see whether b_j is deleted. By Lemma 4, we know that there are $\Omega(N/B^{1-\alpha})$ such b_j's and thus some range query of size N^δ has $\Omega(N^\delta/B^{1-\alpha})$ such b_j's, which is more than the claimed budget $\mathcal{O}(\log_B N + N^\delta/B)$. We note here that the number of d's that are already in memory at the beginning of the range query phase is $M = \mathcal{O}(N^\mu) = o(N^\delta/B^{1-\alpha})$, and thus are insufficient to change the bound. By the Markov inequality, we can say something stronger:

Lemma 6. *There are $\Omega(N^{1-\delta}/B^{1-\alpha})$ range queries of size N^δ so that, to answer any of the queries correctly, any algorithm needs to fetch $\Omega(N^\delta/B^{1-\alpha})$ different d_i's for the required comparisons.*

However, the observation is not sufficient to prove Theorem 2 because a single I/O might fetch back multiple d_i's for the required comparisons. That is the reason why we need CB. Observe further that every such expensive query $\textsc{range}(2, (i-1)N^\delta + 1, iN^\delta)$ needs the existence of some disk page that contain $\Omega(B^\alpha)$ different d_j's for required comparisons, denoted by the set D_i. Note that $c(D_i) \subseteq c(S_{i,N^\delta})$ and therefore $|c(D_i) \cap c(D_j)| \le |c(S_{i,N^\delta}) \cap c(S_{j,N^\delta})| = \mathcal{O}(1)$ for every $i \neq j$. Since there are at most $o(\sqrt{N})$ disk pages containing some d_i, and each of the disk page can be a superset of $\mathcal{O}(B^{1-\alpha})$ different D_i's because $|c(D_i) \cap c(D_j)| = \mathcal{O}(1)$ for $i \neq j$, $B^\alpha > B^{2/3} > \sqrt{B}$ and the following lemma:

Lemma 7. *Let $T_1, T_2, \ldots, T_C$ be the subsets of S, where $|T_i \cap T_j| = \mathcal{O}(1)$ for every $i \neq j \in [C]$ and $|T_i| = \Delta = \omega(\sqrt{|S|})$ for each $i \in [C]$, then $C = \mathcal{O}(|S|/\Delta)$.*

Proof. We prove this by a counting argument. Consider the elements in $D_i = T_i \setminus \bigcup_{j<i} T_j$. Since $|T_i \cap T_j| = \mathcal{O}(1)$ for every $j < i$, $|D_i| = \Delta - \mathcal{O}(i)$. We are done because

$$|S| \ge \sum_{i\in[C]} |D_i| = \sum_{i\in[C]} |T_i| - \mathcal{O}(i) \text{ and thus } C = \mathcal{O}(|S|/\Delta),$$

where the last equality holds due to the fact that $\Delta = \omega(\sqrt{|S|})$. □

The D_i's supply is only $o(N^{1/2}B^{1-\alpha})$. However, the D_i's demand is $\Omega(N^{1-\delta}/B^{1-\alpha})$, implying that some range query requires $\omega(\log_B N + K/B)$ I/Os.

This establishes Theorem 2.

4 Conclusion

In this paper, we consider issues of both practical and theoretical importance in implementations of and algorithms for dictionaries. The development of write optimization has reduced the cost of insertions and deletions. As this tide of insertion/deletion cost recedes, the cost of queries becomes significant in many settings.

We show that natural operations, including count in single dictionaries and delete in compound dictionaries, limit the applicability of write optimization. Our lower bounds correspond to our experience, that these operations do, in fact, mitigate the benefits of write optimization and become bottlenecks of actual systems.

In addition to showing lower bounds that start to put a boundary around the applicability of write optimization and that provide an explanation for the difficulty of implementing fast versions of some operations in databases and file systems, we consider one of our contributions to be the introduction of a set of problems around compound dictionaries, which are a heretofore poorly studied aspects of dictionaries, despite being one of the most common ways in which they are used.

We leave one major open question: can the lower bound for deletes in 2-dictionaries be extended to the entire write-optimization range and raised to show that deletes take $\Omega(\log_B N)$ time, in compound dictionaries with optimal range queries? In other words, can it be shown that each delete requires a search?

References

[1] Apache. HBase. `http://hbase.apache.org`, Last Accessed May 16, 2015, 2015.

[2] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, Feb. 1972.

[3] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. Fogel, B. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 81–92, San Diego, CA, USA, 2007.

[4] M. A. Bender, M. Farach-Colton, M. Goswami, D. Medjedovic, P. Montes, and M. Tsai. The batched predecessor problem in external memory. In A. S. Schulz and D. Wagner, editors, *Proc. 22th Annual European Symposium (ESA)*, pages 112–124, 2014.

[5] G. S. Brodal, E. D. Demaine, J. T. Fineman, J. Iacono, S. Langerman, and J. I. Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1448–1456, 2010.

[6] 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, Baltimore, MD, 2003.

[7] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 859–860, San Francisco, CA, 2000.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] M. Dewey. *Decimal Classification and Relative Index for Libraries, Clippings, Notes, Etc*. Library bureau, 1891.

[10] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. The TokuFS streaming file system. In *Proc. 4th USENIX Workshop on Hot Topics in Storage (HotStorage)*, Boston, MA, USA, June 2012.

[11] Google, Inc. LevelDB: A fast and lightweight key/value database library by Google. `http://github.com/leveldb/`, Last Accessed May 16, 2015, 2015.

[12] D. Heller-Roazen. Tradition's destruction: On the library of alexandria. *October*, 100:133–153, 2002.

[13] J. Hopkins. The 1791 french cataloging code and the origins of the card catalog. *Libraries & Culture*, pages 378–404, 1992.

[14] G. W. Houston. *Inside Roman Libraries: Book Collections and Their Management in Antiquity*. UNC Press Books, 2014.

[15] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. Betrfs: A right-optimized write-optimized file system. In J. Schindler and E. Zadok, editors, *Proc/ 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, February 2015.

[16] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. Betrfs: Write-optimization in a kernel file system. *Transactions on Storage*, 2015. To appear.

[17] W. Johnson, Oct. 2015. Personal communication.

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

[19] L. of Congress. Library of congress classification outline, 2015. Viewed November 8, 2015.

[20] P. O'Neil, E. Cheng, D. Gawlic, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[21] K. Ren and G. A. Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *USENIX Annual Technical Conference*, pages 145–156, 2013.

[22] M. Rouse, Oct. 2015. Personal communication.

[23] R.-H. Rouse. The early library of the sorbonne. *Scriptorium*, 21(1):42–71, 1967.

[24] Tokutek, Inc. TokuDB: MySQL Performance, MariaDB Performance . `http://www.tokutek.com/products/tokudb-for-mysql/`.

[25] Tokutek, Inc. TokuMX—MongoDB Performance Engine. `http://www.tokutek.com/products/tokumx-for-mongodb/`.

[26] Tokutek, Inc. TokuDB and TokuMX, 2014. `http://www.tokutek.com`.

[27] F. J. Witty. The pínakes of callimachus. *The Library Quarterly: Information, Community, Policy*, 28(2):132–136, 1958.

[28] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. Optimizing every operation in a write-optimized file system. In *Proc. 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, Santa Clara, CA, February 2016.