Dynamic Indexability and the Optimality of B-trees[*]

Ke Yi

Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong, China
yike@cse.ust.hk

Abstract

One-dimensional range queries, as one of the most basic type of queries in databases, have been studied extensively in the literature. For large databases, the goal is to build an external index that is optimized for disk block accesses (or I/Os). The problem is well understood in the static case. Theoretically, there exists an index of linear size that can answer a range query in $O(1 + \frac{K}{B})$ I/Os, where K is the output size and B is the disk block size, but it is highly impractical. In practice, the standard solution is the B-tree, which answers a query in $O(\log_B \frac{N}{M} + \frac{K}{B})$ I/Os on a data set of size N, where M is the main memory size. For typical values of N,M, and B, $\log_B \frac{N}{M}$ can be considered a constant.

However, the problem is still wide open in the dynamic setting, when insertions and deletions of records are to be supported. With smart buffering, it is possible to speed up updates significantly to $o(1)$ I/Os amortized. Indeed, several dynamic B-trees have been proposed, but they all cause certain levels of degradation in the query performance, with the most interesting tradeoff point at $O(\frac{1}{B}\log \frac{N}{M})$ I/Os for updates and $O(\log \frac{N}{M} + \frac{K}{B})$ I/Os for queries. In this paper, we prove that the query-update tradeoffs of all the known dynamic B-trees are optimal, when $\log_B \frac{N}{M}$ is a constant. This implies that one should not hope for substantially better solutions for all practical values of the parameters. Our lower bounds hold in a dynamic version of the *indexability model*, which is of independent interests. Dynamic indexability is a clean yet powerful model for studying dynamic indexing problems, and can potentially lead to more interesting lower bound results.

[*]A preliminary version of the paper appeared in *PODS'09*.

1 Introduction

The *B-tree* [7] is a fundamental external index structure used in nearly all database systems. It has both very good space utilization and query performance. External indexes are usually analyzed in the standard *I/O model* [1]. The model parameters include N, the size of the data set to be stored in the index, M, the internal memory size, and B, the disk block size. All of them are measured in terms of data records. In this model, internal computation is free, and we only measure the number of blocks accessed, simply called *I/O*s, during an operation. In the I/O model, the B-tree occupies $O(\frac{N}{B})$ disk blocks, and supports *range reporting queries* in $O(\log_B N + \frac{K}{B})$ I/Os where K is the output size. More precisely, supposing each data record is associated with a *key* from a totally ordered universe, a range (reporting) query retrieves all the data records whose keys are in a given range. Due to the large fanout of the B-tree, for most practical values of N and B, the B-tree is very shallow and $\log_B N$ is essentially a constant. In addition, we can store the top $\Theta(\log_B M)$ levels of the B-tree in the internal memory, further lowering its effective height to $O(\log_B \frac{N}{M})$ (we assume $N > MB$ so that this is always positive). In practice, this means that we can usually get to the desired leaf with merely two or three I/Os, and then start pulling out results.

1.1 Dynamic B-trees

If one wants to update the B-tree directly on disk, it is also well known that it takes $O(\log_B N)$ I/Os. Things become much more interesting if we make use of the internal memory buffer to collect a number of updates and then perform the updates in batches, lowering the amortized update cost significantly. For now let us focus on insertions only; deletions are in general much less frequent than insertions, and there are some generic methods for dealing with deletions by converting them into insertions of "delete signals" [4, 21]. The idea of using a buffer space to batch up insertions has been well exploited in the literature, especially for the purpose of managing archival data, like transaction records and various logs, where there are much more insertions than queries.

The *LSM-tree* [21] was the first along this line of research, by applying the *logarithmic method* [10] to the B-tree. Fix a parameter $2 \le \ell \le B$. It builds a collection of B-trees of sizes up to $M, \ell M, \ell^2 M, \ldots$, respectively, where the first one always resides in memory. An insertion always goes to the memory-resident tree; if the first i trees are full, they are merged together with the $(i+1)$-th tree. If M is large enough, we can do a $(i+1)$-way merge in linear I/Os; if M is so small that only two-way merges are allowed, we can merge them from small to large, the total cost is still linear as the sizes of the B-trees are geometrically increasing. Standard analysis shows that the amortized insertion cost is $O(\frac{\ell}{B}\log_\ell \frac{N}{M})$. A range query takes $O(\log_B N \log_\ell \frac{N}{M} + \frac{K}{B})$ I/Os since $O(\log_\ell \frac{N}{M})$ trees need to be queried. Using *fractional cascading* [13], the query cost can be improved to $O(\log_\ell \frac{N}{M} + \frac{K}{B})$ without affecting the (asymptotic) size of the index and the update cost [9].

Later Jermaine et al. [18] proposed the *Y-tree* also for the purpose of lowering the insertion cost. The Y-tree can be seen as a variant of the *buffer tree* [4]. It is an ℓ-ary tree, where each internal node is associated with a bucket storing all the elements to be pushed down to its subtree. The bucket is emptied only when it has accumulated $\Omega(B)$ elements. Although [18] did not give a formal analysis, it is not difficult to derive that its insertion cost is $O(\frac{\ell}{B}\log_\ell \frac{N}{M})$ and query cost $O(\log_\ell \frac{N}{M} + \frac{K}{B})$, namely, the same as those of the LSM-tree with fractional cascading. Around the same time Buchsbaum et al. [12] independently proposed the *buffered repository tree* in a different context, with similar ideas and the same bounds as the Y-tree.

In order to support even faster insertions, Jagadish et al. [17] proposed the *stepped merge tree*, a variant of the LSM-tree. At each level, instead of keeping one tree of size $\ell^i M$, they keep up to ℓ individual trees. When there are ℓ level-i trees, they are merged to form a level-$(i+1)$ tree (assuming $M > \ell B$). The stepped

	query	insertion
LSM-tree [21] with fractional cascading Y-tree [18] buffered repository tree [12]	$O\left(\log_\ell \frac{N}{M} + \frac{K}{B}\right)$	$O\left(\frac{\ell}{B} \log_\ell \frac{N}{M}\right)$
stepped merge tree [17] with fractional cascading	$O\left(\ell \log_\ell \frac{N}{M} + \frac{K}{B}\right)$	$O\left(\frac{1}{B} \log_\ell \frac{N}{M}\right)$

Table 1: Query/insertion upper bounds of previously known B-tree indexes, for a parameter $2 \le \ell \le B$.

merge tree has an insertion cost of $O(\frac{1}{B} \log_\ell \frac{N}{M})$, lower than that of the LSM-tree. But the query cost is a lot worse, reaching $O(\ell \log_B N \log_\ell \frac{N}{M} + \frac{K}{B})$ I/Os since ℓ trees need to be queried at each level. Again the query cost can be improved to $O(\ell \log_\ell \frac{N}{M} + \frac{K}{B})$ using fractional cascading.

The current best known results are summarized in Table 1. Typically ℓ is set to be a constant [17, 18, 21], at which point all the indexes have the same asymptotic performance of $O(\log \frac{N}{M} + \frac{K}{B})$[1] query and $O(\frac{1}{B} \log \frac{N}{M})$ insertion. Note that the amortized insertion cost of these dynamic B-trees is usually a lot smaller than one I/O, hence much faster than updating the B-tree directly on disk. The query cost is, however, substantially worse than the $O(\log_B \frac{N}{M})$ query cost of the static B-tree by an $\Theta(\log B)$ factor (ignoring the output term $O(K/B)$ which must always exist). As typical values of B range from hundreds to thousands, we are expecting a 10-fold degradation in query performance for these dynamic B-trees. Thus the obvious question is, can we lower the query cost while still allowing for fast insertions?

In particular, the B-tree variants listed in Table 1 are all quite practical ([17, 18, 21] all presented experimental results), so one may wonder if there are some fancy complicated theoretical structures with better bounds that have not been found yet. For the static range query problem, it turned out to be indeed the case. A somehow surprising result by Alstrup et al. [2] shows that it is possible to achieve linear size and $O(K)$ query time in the RAM model. This result also carries over to external memory, yielding a disk-based index with $O(\frac{N}{B})$ blocks and $O(1 + \frac{K}{B})$-I/O query cost. However, this structure is exceedingly complicated, and no one has ever implemented it. In the dynamic case, the *exponential tree* [3] and the dynamic range reporting structure of Mortensen et al. [20] beat the query bounds in Table 1, but their insertion costs are always $\omega(1)$, no matter how large B is. On the other hand, the insertion cost in Table 1 is less than 1 as long as $B \gg \log \frac{N}{M}$ (say, for constant ℓ). Until today no one has improved the tradeoff in Table 1, for B sufficiently large.

1.2 Known lower bounds

Lower bounds for this and related problems have also been sought for. For lower bounds we will only consider insertions; the results certainly also hold for the more general case where insertions and deletions are both present. A closely related problem to range queries is the *predecessor* problem, in which the index stores a set of keys, and the query asks for the preceding key for a query key. The predecessor problem has been extensively studied in various internal memory models, and the bounds are now tight in almost all cases [8]. In external memory, Brodal and Fagerberg [11] proved several tradeoffs between the insertion cost and the query cost for the dynamic predecessor problem. Their lower bound model is a comparison based external memory model. However, a closer look at their proof reveals that one of their techniques can actually be adapted to prove the following tradeoff for range queries for any $B = \omega(1)$: If an insertion takes amortized u I/Os and a query takes worst-case $q + O(\frac{K}{B})$ I/Os, then we have

$$q \cdot \log(uB \log^2 \frac{N}{M}) = \Omega(\log \frac{N}{M}), \tag{1}$$

[1] We use $\log x$ to denote $\log_e x$ in this paper.

provided $u \le 1/\log^3 N$ and $N \ge M^2$. For the most interesting case when we require $q = O(\log \frac{N}{M})$, (1) gives a meaningless bound of $u = \Omega(1/B \log^2 \frac{N}{M})$, as $u \ge 1/B$ trivially. In the other direction, if $u = O(\frac{1}{B} \log \frac{N}{M})$, the tradeoff (1) still leaves an $\Theta(\log \log \frac{N}{M})$ gap to the known upper bound for q.

1.3 Our results

In this paper, we prove a query-insertion tradeoff of

$$u = \begin{cases} \Omega\left(\frac{q}{B} B^{1/q}\right), & \text{for } q < \alpha \log B, \text{any constant } \alpha; \\ \Omega\left(\frac{\log B}{B \log q}\right), & \text{for all } q, \end{cases} \tag{2}$$

for any dynamic range query index with a query cost of $q + O(K/B)$ and an amortized insertion cost of u, provided $N \ge 2MB^2$. For most reasonable values of N, M, and B, we may assume that $\frac{N}{M} = B^{O(1)}$, or equivalently that the B-tree built on N keys has constant height. Let us first consider the following three most interesting tradeoff points:

(i) If we require $q = O(\log \frac{N}{M}) = O(\log B)$, the first branch of (2) gives $u = \Omega(\frac{1}{B} \log B)$, matching the known upper bounds in Table 1 for constant ℓ. In the other direction, if $u = O(\frac{1}{B} \log \frac{N}{M}) = O(\frac{1}{B} \log B)$, we have $q = \Omega(\log B) = \Omega(\log \frac{N}{M})$, which is again tight.

(ii) If we want constant query cost $q = O(1)$, then the first branch of (2) gives $u = \frac{1}{B} B^{\Omega(1)}$, an exponential blowup from case (i). This also matches the first row of Table 1 for $\ell = B^{\Theta(1)}$.

(iii) If we want the smallest insertion cost $u = O(\frac{1}{B})$, the second branch of (2) yields $q = B^{\Omega(1)}$, again an exponential blowup from case (i), and matches the second row of Table 1 for $\ell = B^{\Theta(1)}$.

Rewriting the upper bounds in Table 1 will better reveal the tightness of the tradeoff. For the first row, from $q = O(\log_\ell B)$ we get $\ell = B^{O(1/q)}$, so $u = O(\frac{\ell}{B} \log_\ell B) = O(\frac{q}{B} B^{O(1/q)})$, matching the first branch of (2). For the second row, from $q = O(\ell \log_\ell B)$ we get $\ell = \Omega(\frac{q}{\log B} \log \frac{q}{\log B})$, so $u = O(\frac{1}{B} \log_\ell B) = O\left(\frac{\log B}{B \log(q/\log B)}\right)$, which matches the second branch of (2) except in the narrow range $\omega(\log B) \le q \le o(\log^{1+\epsilon} B)$, for any small constant $\epsilon > 0$, whereas in this range, the gap between the upper and lower bounds is merely $\Theta\left(\frac{\log q}{\log(q/\log B)}\right) = o(\log \log B)$. Figure 1 pictorially illustrates the entire query-insertion tradeoff for both upper and lower bounds.

Our results establish the optimality of dynamic B-trees in the parameter range $\frac{N}{M} = B^{O(1)}$. This, however, may raise the theoretical question whether they are optimal for all values of N, ideally unbounded in terms of B. The answer is no, unfortunately. We will discuss more on this issue and outline some right open questions to ask towards the end of the paper.

We obtain our lower bounds in a model which we call *dynamic indexability*. It is a natural dynamic version of the *indexability model* [14], which was originally proposed by Hellerstein, Koutsoupias, and Papadimitriou [15]. To date, most known lower bounds for indexing problems are proved in this model [5, 6, 14, 19, 22]. Intuitively speaking, this model assumes that the query cost is only determined by the number of disk blocks that hold the actual query results, and ignores all the search cost that we need to pay to find these blocks (see Section 2 for details). Ignoring the search cost may seem too permissive, and indeed the model yields trivial bounds for static one-dimensional problems. In fact, until today this model has been used exclusively for two or higher dimensional problems. In the JACM article [14] that summarizes most of the results on indexability, the authors state: "However, our model also ignores the dynamic aspect of the

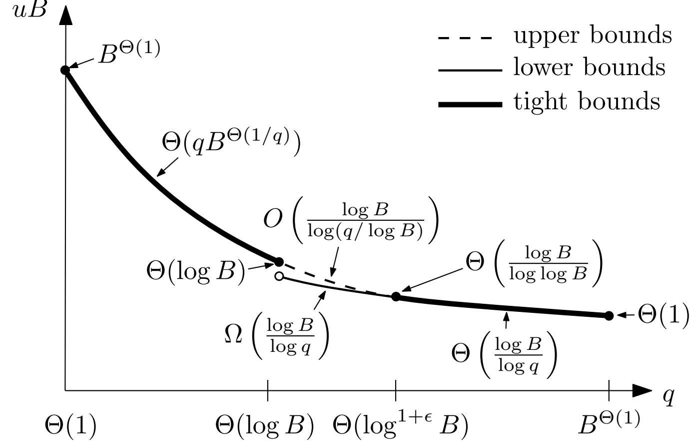

Figure 1: The query-insertion tradeoff, assuming $\log_B \frac{N}{M} = O(1)$. The vertical axis shows uB instead of u to avoid repeating the $\frac{1}{B}$ factors in the figure.

problem, that is, the cost of insertion and deletion. Its consideration *could* be a source of added complexity, and in a more general model the source of more powerful lower bounds." In this respect, another contribution of this paper is to add dynamization to the model of indexability, making it more powerful and complete. In particular, our lower bound results suggest that, although static indexability is useful only in two or more dimensions, dynamization makes it a suitable model for the more basic one-dimensional indexing problems.

As a final remark, our lower bounds actually hold for the simpler *multi-point* queries, where the keys of the data records do not have to be unique, and the query simply retrieves all data records with a given key. It is clear that multi-point queries are special range queries.

2 Dynamic Indexability

2.1 Static indexability

We first review the framework of static indexability before introducing its dynamization. We follow the notation from [14]. A *workload* W is a tuple $W = (D, I, \mathcal{Q})$ where D is the *domain* of all possible elements, $I \subseteq D$ is a finite subset of D (the *instance*), and $\mathcal{Q}$ is a set of subsets of I (the *query set*). For example, for range queries, D is a totally ordered universe of all possible keys (in this model we do not distinguish between an element and its key), I is a set of keys, and $\mathcal{Q}$ consists of $I \cap [x, y]$ for all $x, y \in D$. We usually use $N = |I|$ to denote the number of elements in the instance. An *indexing scheme* $\mathcal{S} = (W, \mathcal{B})$ consists of a workload W and a set $\mathcal{B}$ of B-subsets of I such that $\mathcal{B}$ covers I, i.e., $I \subseteq \bigcup_{\beta \in \mathcal{B}} \beta$. The B-subsets of $\mathcal{B}$ model the data blocks of an index structure, while any auxiliary structures connecting these data blocks (such as pointers, splitting elements) are ignored from this framework. The size of the indexing scheme is $|\mathcal{B}|$, the number of blocks. In [14], an equivalent parameter, the *redundancy* $r = B|\mathcal{B}|/N$ is used to measure the space complexity of the indexing scheme. The cost of a query $Q \in \mathcal{Q}$ is the minimum number of blocks whose union covers Q. Note that here we have implicitly assumed that the query algorithm can find these blocks to cover Q with no cost, essentially ignoring the "search cost". The *access overhead* A is the minimum A such that any query $Q \in \mathcal{Q}$ has a cost at most $A \cdot \lceil |Q|/B \rceil$. Note that $\lceil |Q|/B \rceil$ is the minimum number of blocks to access so as to report all elements in Q, so the access overhead A measures

how much more multiplicatively we need to access in order to retrieve Q. For some problems using a single parameter for the access overhead is not expressive enough, and we may split it into a multiplicative factor plus an additive term. More precisely, an indexing scheme with access overhead (A_0, A_1) must answer any query $Q \in \mathcal{Q}$ with cost at most $A_0 + A_1 \cdot \lceil |Q|/B \rceil$ [6].

Except for some trivial facts, all the lower bound results obtained under this model are expressed as a tradeoff between r and A (or (A_0, A_1)), for some construction of the workload. For example, a two-dimensional range query index has a tradeoff of $r = \Omega(\log(N/B)/\log A)$ [5, 14]; for the 2D *point enclosure* problem, the dual of range queries, we have the tradeoff $A_0 A_1^2 = \Omega(\log(N/B)/\log r)$ [6]. These results show that, even if we ignore the search cost, we can still obtain nontrivial lower bounds for these problems. These lower bounds have also been matched with corresponding indexes that *do* include the search cost, for typical values of r and A [5, 6]. This means that the inherent difficulty for these indexing problems roots from how we should *layout* the data objects on disk, not the search structure on top of them. By ignoring the search component of an index, we obtain a simple and clean model, which is still powerful enough to reveal the inherent complexity of indexing. It should be commented that the indexability model is very similar in spirit to the *cell probe model* of Yao [23], which has been successfully used to derive many internal memory lower bounds. But the two models are also different in some fundamental ways; please see [14] for a discussion.

Nevertheless, although the indexability model is appropriate for two-dimensional problems, it seems to be overly strong for the more basic one-dimensional range query problem. In one dimension, we could simply lay out all the elements in the key order sequentially on disk, which gives us a trivial indexability index with $r = A = 1$! On the other hand, designing an actual index in the I/O model that matches these bounds (up to constant factors) is highly nontrivial [2], and it is quite surprising that this is actually achievable.

2.2 Dynamic indexability

In the dynamic case, the domain D remains static, but the instance set I could change. Correspondingly, the query set $\mathcal{Q}$ changes and the index also updates its blocks $\mathcal{B}$ to cope with the changes in I. In the static model, there is no component to model the main memory, which is all right since the memory does not help reduce the worst-case query cost anyway. However, in the dynamic case, the main memory does improve the update cost significantly by buffering the recent updates, so we have to include a main memory component in the indexing scheme. More precisely, the workload W is defined as before, but an indexing scheme is now defined as $\mathcal{S} = (W, \mathcal{B}, \mathcal{M})$ where $\mathcal{M}$ is a subset of I with size at most M such that the blocks of $\mathcal{B}$ together with $\mathcal{M}$ cover I. The redundancy r is defined as before, but the access overhead A is now defined as the minimum A such that any $Q \in \mathcal{Q}$ can be covered by $\mathcal{M}$ and at most $A \cdot \lceil |Q|/B \rceil$ blocks from $\mathcal{B}$.

We now define the *dynamic indexing scheme*. Here we only consider insertions as our focus is to prove lower bounds. We first define a *dynamic workload*.

Definition 1 A *dynamic workload* $\mathbb{W}$ is a sequence of N workloads $W_1 = (D, I_1, \mathcal{Q}_1), \ldots, W_N = (D, I_N, \mathcal{Q}_N)$ such that $|I_i| = i$ and $I_i \subset I_{i+1}$ for $i = 1, \ldots, N-1$.

Essentially, we insert N elements into I one by one, resulting in a sequence of workloads. Meanwhile, the query set $\mathcal{Q}$ changes according to the problem at hand.

Definition 2 Given a dynamic workload $\mathbb{W} = (W_1, \ldots, W_N)$, a *dynamic indexing scheme* $\mathbb{S}$ is a sequence of N indexing schemes $\mathcal{S}_1 = (W_1, \mathcal{B}_1, \mathcal{M}_1), \ldots, \mathcal{S}_N = (W_N, \mathcal{B}_N, \mathcal{M}_N)$. Each $\mathcal{S}_i$ is called a *snapshot* of

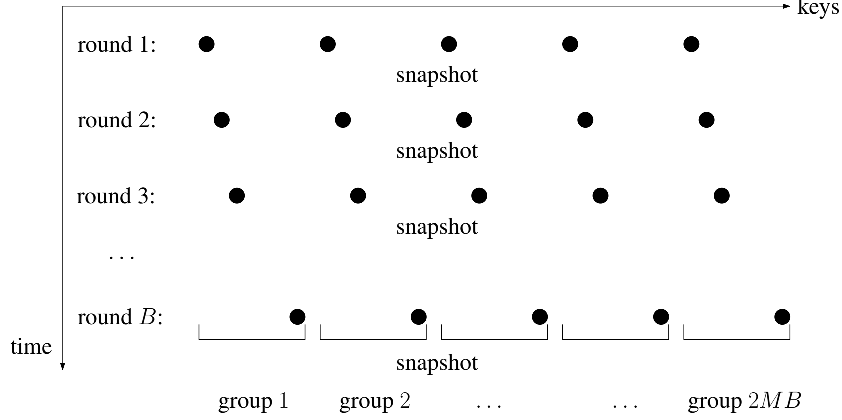

Figure 2: Construction of the workload.

$\mathbb{S}$. We say that $\mathbb{S}$ has redundancy r and access overhead A if for all i, $\mathcal{S}_i$ has redundancy at most r and access overhead at most A.

A third parameter u, the *update cost*, is defined as follows.

Definition 3 Given a dynamic indexing scheme $\mathbb{S}$, the *update cost* from $\mathcal{S}_i$ to $\mathcal{S}_{i+1}$ is $|\mathcal{B}_{i+1} - \mathcal{B}_i|$, i.e., the number of new blocks created in $\mathcal{B}_{i+1}$. The (amortized) *update cost* of $\mathbb{S}$ is the minimum u such that the sum of all the update costs from $\mathcal{S}_i$ to $\mathcal{S}_{i+1}$ for all $0 \leq i < N$ is uN, where we set $\mathcal{B}_0 = \emptyset$.

Note that the definition above ignores the cost of deleting blocks in $\mathcal{B}_i$. Indeed, the indexing scheme $\mathbb{S}$ does not need to delete any blocks, if the update cost is the main concern. In any case, considering both the cost of creating blocks and removing blocks will not affect u by more than a factor of 2, so we choose the definition above to simplify the model.

Our definition of the dynamic indexability model maintains the spirit of the static case: we will only focus on the cost associated with the changes in the blocks holding the actual data objects, while ignoring the search cost of finding these blocks and deciding how they should be changed.

Two extreme cases for range query indexes. As we will see shortly, all our lower bounds are expressed as tradeoffs between the access overhead A and the update cost u; the redundancy r turns out to be irrelevant. Considering the following two extreme cases for a range query index will give us some intuitive ideas on the tradeoff between A and u.

We will use the workload construction as illustrated in Figure 2 with $N = 2MB^2$ keys. The keys are divided into $2MB$ groups of B each so that they are consecutive within every group but non-overlapping across different groups. We perform the insertions in B rounds; in each round, we simply add one key to each group. The dynamic indexing scheme $\mathbb{S}$ correspondingly has N snapshots $\mathcal{S}_1 = (W_1, \mathcal{B}_1, \mathcal{M}_1), \ldots, \mathcal{S}_N = (W_N, \mathcal{B}_N, \mathcal{M}_N)$. We will only consider the subsequence $\mathbb{S}'$ consisting of the snapshots $\mathcal{S}_{2MB}, \mathcal{S}_{2\cdot 2MB}, \ldots, \mathcal{S}_N$, i.e., those after every round. The total update cost of this subsequence is obviously no higher than that of the entire sequence. We will only consider queries that report one group of at most B keys.

The first extreme case is the minimum possible update cost $u = 1/B$. In this case, $\mathbb{S}'$ is only allowed to add one block for every B insertions. In every round, all groups each have a new key arriving. Since the memory has size M, at least $2MB - M$ groups will have their new keys written in newly created blocks. These blocks do not contain any keys from the previous rounds. Thus after B rounds, there are at least

$2MB - MB = MB$ groups each of which is completely shattered: the B keys are all in different blocks. Querying these groups requires an access overhead of $A = B$ — the worst possible access overhead.

The second extreme case is minimum possible access overhead $A = 1$. In this case, in any snapshot of $\mathbb{S}'$, all the keys in every group must reside in one block, and possibly also in memory. However, since the memory has size M and there are $2MB$ groups, at least MB of them must have all their keys on disk in all the snapshots of $\mathbb{S}'$. This means that in every snapshot of $\mathbb{S}'$, a new block is created for each of these groups. Thus the total update cost is MB^2, which means that $u \geq 1/2$. One can easily push this to $u \geq 1-\epsilon$ for any constant $\epsilon > 0$ using a larger N. This is almost tight. For an upper bound, we can simply store all the keys in order such that each block contains between $B/2$ and B keys. To insert a new key, we just add it to the corresponding block; if the block already contains B keys, we split it into 2 blocks. Since a split happens once every $\Omega(B)$ insertions, the amortized update cost is $u = 1 + O(\frac{1}{B})$. The redundancy of this simple scheme is $A = 2$.

3 Lower Bounds

We prove the following tradeoff for any range query index under the dynamic indexability model.

Theorem 1 *Let $\mathbb{S}$ be any dynamic indexing scheme for range queries with access overhead A and update cost u. Provided $N \geq 2MB^2$, we have*

$$u = \begin{cases} \Omega\left(\frac{A}{B} \cdot B^{1/A}\right), & \text{for } A < \alpha \log B, \text{any constant } \alpha; \\ \Omega\left(\frac{\log B}{B \log A}\right), & \text{for all } A. \end{cases}$$

Note that this lower bound does not depend on the redundancy r, meaning that the index cannot do better by consuming more space. Since a query bound of $q + O(K/B)$ implies that any query can be answered by accessing $O(q \cdot \lceil K/B \rceil)$ blocks, we can replace A with q in the theorem and get the tradeoff in (2), but bear in mind that Theorem 1 is actually stronger than (2).

To prove Theorem 1, below we first define a *ball-shuffling* problem and show that any dynamic indexing scheme for range queries yields a solution to the ball-shuffling problem. Then we prove a lower bound for the latter.

3.1 The ball-shuffling problem and the reduction

We now define the *ball-shuffling* problem, and present a lower bound for it. There are b balls and t bins, $\beta_1, \ldots, \beta_t$. The balls come one by one. Upon the arrival of each ball, we need to find some bin β_i to put it in. We use $|\beta_i|$ to denote the current number of balls in β_i. The *cost* of putting the ball into β_i is defined to be $|\beta_i| + 1$. Instead of directly putting a ball into a bin, we can do so with *shuffling*: first collect all the balls from one or more bins, add the new ball to the collection, and then arbitrarily allocate these balls into a number of empty bins. The cost of this operation is the total number of balls involved, i.e., if J denotes the set of indices of the bins collected, the cost is $\sum_{i \in J} |\beta_i| + 1$. Note that directly putting a ball into a bin can be seen as a special shuffle, where we collect balls from only one bin and allocate the balls back to one bin.

Our main result for the ball-shuffling problem is the following lower bound, whose proof is deferred to Section 3.2.

Theorem 2 *The cost of any algorithm for the ball-shuffling problem is at least*
(i) $\Omega(tb^{1+1/t})$ for $t < \alpha \log b$ where α is an arbitrary constant; and
(ii) $\Omega(b \log_t b)$ for any t.

The reduction. Suppose there is a dynamic indexing scheme $\mathbb{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_N)$ for range queries with update cost u and access overhead A. Assuming $N \geq 2MB^2$, we will show how this leads to a solution to the ball-shuffling problem on $b = B$ balls and $t = A$ bins with cost $O(uB^2)$. This will immediately translate the tradeoff in Theorem 2 to the desired tradeoff in Theorem 1.

We divide the insertions of N elements into batches of $2MB^2$, and for each batch, we use the construction in Figure 2. Since the amortized cost for handling every insertion is u, at least one of the batches has a total update cost of at most $O(uMB^2)$. We will just focus on one such batch. As before, we only consider the subsequence $\mathbb{S}'$ of B snapshots after each round in the batch. Let $\mathcal{B}_i$ be the collection of blocks after round $i, i = 1, \ldots, B$. Note that in the construction of Figure 2, we only consider range queries that retrieve a group, they are just multi-point queries if we use the same key for all elements in each group. Thus, our lower bounds will hold for multi-point queries as well.

Recall that the update cost from a snapshot $\mathcal{S} = (W, \mathcal{B}, \mathcal{M})$ to its succeeding snapshot $\mathcal{S}' = (W', \mathcal{B}', \mathcal{M}')$ is the number of blocks that are new in $\mathcal{B}'$ compared with $\mathcal{B}$. We now define the *element update cost* to be the number of elements in these new blocks, i.e., $\sum_{\beta \in \mathcal{B}' - \mathcal{B}} |\beta|$. Since each block contains at most B elements, the element update cost is at most B times larger than the update cost. Thus, $\mathbb{S}'$ has an element update cost of $O(uMB^3)$. The element update cost can be associated with the elements involved, that is, it is the total number of times that an element has been in a newly created block, summed over all elements.

If a group G has at least one element appearing in main memory in some snapshot of $\mathbb{S}'$, then it is said to be *contaminated*. Since at most MB elements can ever reside in main memory in the B snapshots, at most MB groups are contaminated. Since the total element update cost of $\mathbb{S}'$ is $O(uMB^3)$, among the at least MB uncontaminated groups, at least one group, say G, has an element update cost of $O(uB^2)$. Note that the element update cost of G over the whole sequence is $\sum_{i=1}^{B-1} \sum_{\beta \in \mathcal{B}_{i+1} - \mathcal{B}_i} |\beta \cap G|$. Let $G_1, \ldots, G_B$ be the sets of elements of this group after every round. Since this group is uncontaminated, all elements in G_i must be completely covered by $\mathcal{B}_{i \cdot 2MB}$ for each $i = 1, \ldots, B$. Since G_i has at most B elements and $\mathbb{S}$ has access overhead A, G_i should always be covered by at most A blocks in $\mathcal{B}_{i \cdot 2MB}$. For each i, let $\beta_{i,1}, \ldots, \beta_{i,A}$ be the blocks of $\mathcal{B}_{i \cdot 2MB}$ that cover G_i, let $\hat{\beta}_{i,j} = \beta_{i,j} \cap G, j = 1, \ldots, A$. Note that these $\hat{\beta}_{i,j}$'s may overlap and some of them may be empty. Let $\hat{\mathcal{B}}_i = \{\hat{\beta}_{i,1}, \ldots, \hat{\beta}_{i,A}\}$. Consider the transition from $\hat{\mathcal{B}}_i$ to $\hat{\mathcal{B}}_{i+1}$. We can as before define its element update cost as $\sum_{\hat{\beta} \in \hat{\mathcal{B}}_{i+1} - \hat{\mathcal{B}}_i} |\hat{\beta}|$. Let us compare this element update cost with the actual element update cost from $\mathcal{B}_{i \cdot 2MB}$ to $\mathcal{B}_{(i+1) \cdot 2MB}$ for the elements of G, i.e., $\sum_{\beta \in \mathcal{B}_{i+1} - \mathcal{B}_i} |\beta \cap G|$. We see that the former is no higher than the latter: We count an element in $\sum_{\hat{\beta} \in \hat{\mathcal{B}}_{i+1} - \hat{\mathcal{B}}_i} |\hat{\beta}|$ if it resides in a new block $\hat{\beta} \in \hat{\mathcal{B}}_{i+1}$. In this case, we must have also counted it in $\sum_{\beta \in \mathcal{B}_{i+1} - \mathcal{B}_i} |\beta \cap G|$, because if $\hat{\beta}$ is new, β must be new as well. Therefore, the total element update cost of the sequence $\hat{\mathcal{B}}_1, \ldots, \hat{\mathcal{B}}_B$ is at most $O(uB^2)$.

Now we claim that the sequence $\hat{\mathcal{B}}_1, \ldots, \hat{\mathcal{B}}_B$ gives us a solution to the ball-shuffling problem of B balls and A bins with cost at most its element update cost. Indeed, if we treat the blocks in $\hat{\mathcal{B}}_i$ as bins and the B elements in the group as balls, this is almost exactly the ball-shuffling problem, with the element update cost from $\hat{\mathcal{B}}_i$ to $\hat{\mathcal{B}}_{i+1}$ equal to the shuffling cost of adding the $(i+1)$-st ball. One technical issue is that one element may appear in multiple blocks in a $\hat{\mathcal{B}}_i$ as we allow redundancy in the indexing scheme. Below we give a more precise construction of a solution to the ball-shuffling problem from $\hat{\mathcal{B}}_1, \ldots, \hat{\mathcal{B}}_B$ that takes this issue into consideration. Again suppose $\hat{\beta}_{i,1}, \ldots, \hat{\beta}_{i,A}$ are the blocks in $\hat{\mathcal{B}}_i$. Let $\beta'_{i,1}, \ldots, \beta'_{i,A}$ be the bins after the i-th element (or equivalently, the i-th ball) has been added. In the construction we will maintain the invariant that $\beta'_{i,j} \subseteq \hat{\beta}_{i,j}$ for all j and $\beta'_{i,j_1} \cap \beta'_{i,j_2} = \emptyset$ for $j_1 \neq j_2$. The second condition ensures that elements are not duplicated. At $i = 1$, we put the first ball into any $\beta'_{1,j}$ where $\hat{\beta}_{1,j}$ is not empty, establishing the initial invariant. As we go from $\hat{\mathcal{B}}_i$ to $\hat{\mathcal{B}}_{i+1}$, suppose without loss of generality that the first k blocks

change, i.e., $\hat{\beta}_{i,1}, \ldots, \hat{\beta}_{i,k}$ change to $\hat{\beta}_{i+1,1}, \ldots, \hat{\beta}_{i+1,k}$, while the remaining $A-k$ blocks remain the same. We will correspondingly shuffle the bins $\beta'_{i,1}, \ldots, \beta'_{i,k}$, and allocate the balls following $\hat{\beta}_{i+1,1}, \ldots, \hat{\beta}_{i+1,k}$, ensuring that $\beta'_{i+1,j} \subseteq \hat{\beta}_{i+1,j}$, while removing all the duplicated balls. More precisely, we use the following allocation procedure: Let $R = \beta'_{i+1,k} \cup \cdots \cup \beta'_{i+1,A}$ be the set of balls in the other $A-k$ bins that are not changing. For each of $j = 1, \ldots, k$ in order, we set $\beta'_{i+1,j} = \hat{\beta}_{i+1,j} - R - \beta'_{i+1,1} - \cdots - \beta'_{i+1,j-1}$. It is clear that the invariant above is maintained after this allocation procedure. Also note that since the new ball must appear in at least one of the $\hat{\beta}_{i+1,j}$'s, $j = 1, \ldots, k$, and must not appear in R, this procedure will allocate it in one of the bins $\beta'_{i+1,j}$, so this indeed leads to a valid solution to the ball-shuffling problem. Finally, by the invariant $\beta'_{i+1,j} \subseteq \hat{\beta}_{i+1,j}$, the cost of this shuffle must be no larger than the element update cost form $\hat{\mathcal{B}}_i$ to $\hat{\mathcal{B}}_{i+1}$. Therefore, we obtain a solution to the ball-shuffling problem with cost $O(uB^2)$. This completes the reduction.

Tightness of the bounds. By the above reduction and the tightness of our lower bounds (2) as illustrated in Figure 1, the lower bounds for the ball shuffling problem in Theorem 2 must also be tight. Through the reduction, one can certainly convert the existing dynamic B-trees to ball-shuffling solutions that match the lower bounds, but this will be painful and non-illustrative. Below we give direct and much simpler solutions to the ball-shuffling problem, which hopefully gives us more insight into the problem itself.

For $t \geq 2\log b$, we use the following shuffling strategy. Let $x = t/\log b \geq 2$. Divide the t bins evenly into $\log_x b$ groups of $t/\log_x b$ each. We use the first group to accommodate the first $t/\log_x b$ balls. Then we shuffle these balls to one bin in the second group. In general, when all the bins in group i are occupied, we shuffle all the balls in group i to one bin in group $i+1$. The total cost of this algorithm is obviously $b\log_x b$ since each ball has been shuffled $\log_x b$ times. To show that this algorithm actually works, we need to show that all the b balls can be indeed accommodated. Since the capacity of each group increases by a factor of $t/\log_x b$, the capacity of the last group is

$$\left(\frac{t}{\log_x b}\right)^{\log_x b} = \left(\frac{xt}{x\log_x b}\right)^{\log_x b} = b\left(\frac{t}{x\log_x b}\right)^{\log_x b} = b\left(\frac{\log b}{\log_x b}\right)^{\log_x b} = b(\log x)^{\log_x b} \geq b.$$

This corresponds to the upper bound curve in Figure 1 from $q = \Theta(\log B)$ to $q = B^{\Theta(1)}$.

Part (i) of the theorem concerns with $t = O(\log b)$. For these small values of t we need to deploy a different strategy. We always put balls one by one to the first bin β_1. When β_1 has collected $b^{1/t}$ balls, we shuffle all the balls to β_2. Afterward, every time β_1 reaches $b^{1/t}$, we merge all the balls in β_1 and β_2 and put the balls back to β_2. For β_2, every time it has collected $b^{2/t}$ balls from β_1, we merge all the balls with β_3. In general, every time β_i has collected $b^{i/t}$ balls, we move all the balls to β_{i+1}. Let us compute the total cost of this strategy. For each shuffle, we charge its cost to the destination bin. Thus, the cost charged to β_1 is at most $(b^{1/t})^2 \cdot b^{1-1/t} = b^{1+1/t}$, since for every batch of $b^{1/t}$ balls, it pays a cost of at most $(b^{1/t})^2$ to add them one by one, and there are $b^{1-1/t}$ such batches. In general, for any bin $\beta_i, 1 \leq i \leq t$, the balls arrive in batches of $b^{(i-1)/t}$, and the bin clears itself after every $b^{1/t}$ such batches. The cost for each batch is at most $b^{i/t}$, the maximum size of β_i, so the cost for all the $b^{1/t}$ batches before β_i clears itself is at most $b^{(i+1)/t}$. The bin clears itself $b/b^{i/t} = b^{1-i/t}$ times, so the total cost charged to β_i is $b^{1+1/t}$. Therefore, the total cost charged to all the bins is $tb^{1+1/t}$. This gives us the tight upper bound curve in Figure 1 from $q = \Theta(1)$ to $q = \Theta(\log B)$.

3.2 Proof of Theorem 2

We first make a few useful observations with respect to the ball-shuffling problem. Since as defined the balls are indistinguishable, and so are the bins, then during the ball shuffling process, the current state can be completely characterized by the multiset of bin sizes $S = \{|\beta_1|, \ldots, |\beta_t|\}$. Then during a shuffle, it is unreasonable to create a bin with size the same as one of the bins this shuffle has collected, if one wants to minimize the shuffling cost. Realizing this, if the sizes of the bins before and after the shuffle are S and S', respectively, then the cost of the shuffle can be effectively written as $\sum(S' - S)$, where "$-$" should be understood as (multi)set difference taking into account the multiplicities, and "$\sum X$" denotes the summation of the numbers in the multiset X.

Proof of part (ii). We first prove the easier part of the theorem, that the ball-shuffling cost with b balls and t bins is at least $\Omega(b \log_t b)$. We will take an indirect approach, proving that any algorithm that handles the balls with an average cost of u using t bins cannot accommodate $(2t)^{\lceil 2u \rceil}$ balls or more. This means that $b < (2t)^{\lceil 2u \rceil}$, or $u = \Omega(\log_t b)$, so the total cost of the algorithm is $ub = \Omega(b \log_t b)$.

We prove that $b < (2t)^{2u}$ by induction for all $u \geq 1$ that is a multiple of $\frac{1}{2}$; this suffices for showing $b < (2t)^{\lceil 2u \rceil}$ for all values of $u \geq 1$. When $u = 1$, clearly the algorithm has to put every ball into an empty bin, so with t bins, the algorithm can handle at most $t < (2t)^2$ balls. Now assume that the claim is true for u, and we will show that it is also true for $u + \frac{1}{2}$. Equivalently we need to show that to handle $(2t)^{2u+1}$ balls, any algorithm using t bins has to pay an average cost of more than $u + \frac{1}{2}$ per ball, or $(u + \frac{1}{2})(2t)^{2u+1} = (2tu + t)(2t)^{2u}$ in total. We divide the $(2t)^{2u+1}$ balls into $2t$ batches of $(2t)^{2u}$ each. By the induction hypothesis, to handle the first batch, the algorithm has to pay a total cost of more than $u(2t)^{2u}$. For each of the remaining batches, the cost is also more than $u(2t)^{2u}$, plus the cost of shuffling the existing balls from previous batches. This amounts to a total cost of $2tu(2t)^{2u}$, and we only need to show that shuffling the balls from previous batches costs at least $t(2t)^{2u}$ in total.

If a batch has at least one ball that is never shuffled in later batches, it is said to be a *bad batch*, otherwise it is a *good batch*. The claim is that at most t of these $2t$ batches are bad. Indeed, since each bad batch has at least one ball that is never shuffled later, the bin that this ball resides in cannot be touched any more. So each bad batch takes away at least one bin from later batches and there are only t bins. Therefore there are at least t good batches, in each of which all the $(2t)^{2u}$ ball have been shuffled later. This costs at least $t(2t)^{2u}$, and the proof completes.

The merging lemma. Part (ii) of the theorem is very loose for small values of t. If $t \leq \alpha \log b$ where α is any constant, we can prove a much higher lower bound $\Omega(tb^{1+1/t})$, which later will lead to the more interesting branch in the query-update tradeoff (2) of range queries. The rest of this section is devoted to the proof of part (i) of Theorem 2. Note that it is important to prove the lower bound for an arbitrary constant α, as there could be any hidden constant in the $O(\log B)$ query upper bound.

We first prove the following lemma, which restricts the way how an optimal algorithm might do shuffling. We call a shuffle that allocates balls back to more than one bin a *splitting* shuffle, otherwise it is a *merging* shuffle.

Lemma 1 *There is an optimal algorithm that only uses merging shuffles.*

Proof: For a shuffle, we call the number of bins that receive balls from the shuffle its *splitting number*. A splitting shuffle has a splitting number at least 2, and a merging shuffle's splitting number is 1. For an algorithm $\mathcal{A}$, let $\pi(\mathcal{A})$ be the sequence of the splitting numbers of all the b shuffles performed by $\mathcal{A}$. Below we will show how to transform $\mathcal{A}$ into another algorithm $\mathcal{A}'$ whose cost is no higher than that of $\mathcal{A}$, while

$\pi(\mathcal{A}')$ is lexicographically smaller than $\pi(\mathcal{A})$. Since every splitting number is between 1 and t, after a finite number of such transformations, we will arrive at an algorithm whose splitting numbers are all 1, hence proving the lemma.

Let $\mathcal{A}$ be an algorithm that uses at least one splitting shuffle, and consider the last splitting shuffle carried out by $\mathcal{A}$. Suppose it allocates balls to k bins. $\mathcal{A}'$ will do the same as $\mathcal{A}$ up until its last splitting shuffle, which $\mathcal{A}'$ will change to the following shuffle. $\mathcal{A}'$ will collect balls from the same bins but will only allocate them to $k-1$ bins. Among the $k-1$ bins, $k-2$ of them receive the same number of balls as in $\mathcal{A}$, while the last bin receives all the balls in the last two bins used in $\mathcal{A}$. So the only difference between $\mathcal{A}$ and $\mathcal{A}'$ after this shuffle is two bins, say β_1, β_2 of $\mathcal{A}$ and β_1', β_2' of $\mathcal{A}'$. Note that the cost of this shuffle is the same for both $\mathcal{A}$ and $\mathcal{A}'$. After this shuffle, suppose we have $|\beta_1| = x, |\beta_2| = y, |\beta_1'| = x+y, |\beta_2'| = 0$ for some $x, y \geq 1$. Clearly, no matter what $\mathcal{A}'$ does in the future, $\pi(\mathcal{A}')$ is always lexicographically smaller than $\pi(\mathcal{A})$.

From now on $\mathcal{A}'$ will mimic what $\mathcal{A}$ does with no higher cost. We will look ahead at the operations that $\mathcal{A}$ does with β_1 and β_2, and decide the corresponding actions of $\mathcal{A}'$. Note that $\mathcal{A}$ will do no more splitting shuffles. Consider all the shuffles that $\mathcal{A}$ does until it merges β_1 and β_2 together, or until the end if $\mathcal{A}$ never does so. For those shuffles that touch neither β_1 nor β_2, $\mathcal{A}'$ will simply do the same. Each of the rest of the shuffles involves β_1 but not β_2 (resp. β_2 but not β_1). Since the bins are indistinguishable, for any such merging shuffle, we may assume that all the balls are put back to β_1 (resp. β_2). Suppose there are a_1 shuffles involving β_1 and a_2 shuffles involving β_2. Assume for now that $a_1 \leq a_2$. $\mathcal{A}'$ will do the following correspondingly. When $\mathcal{A}$ touches β_1, $\mathcal{A}'$ will use β_1'; and when $\mathcal{A}$ touches β_2, $\mathcal{A}'$ will use β_2'. Clearly, for any shuffle that involves neither β_1 nor β_2, the cost is the same for $\mathcal{A}$ and $\mathcal{A}'$. For a shuffle that involves β_1 but not β_2, since before $\mathcal{A}$ merges β_1 and β_2, we have the invariant that $|\beta_1'| = |\beta_1| + y$, $\mathcal{A}'$ pays a cost of y more than that of $\mathcal{A}$, for each of these a_1 shuffles. For a shuffle that involves β_2 but not β_1, since we have the invariant that $|\beta_2'| = |\beta_2| - y$, $\mathcal{A}'$ pays a cost of y less than that of $\mathcal{A}$, for each of these a_2 shuffles. So $\mathcal{A}'$ incurs a total cost no more than that of $\mathcal{A}$. In the case $a_1 \geq a_2$, when $\mathcal{A}$ touches β_1, $\mathcal{A}'$ will use β_2'; and when $\mathcal{A}$ touches β_2, $\mathcal{A}'$ will use β_1'. A similar argument then goes through. Finally, when $\mathcal{A}$ merges β_1 and β_2 together (if it ever does so), $\mathcal{A}'$ will also shuffle both β_1' and β_2'. Since we always have $|\beta_1| + |\beta_2| = |\beta_1'| + |\beta_2'|$, the cost of this shuffle is the same for $\mathcal{A}$ and $\mathcal{A}'$. After this shuffle, $\mathcal{A}$ and $\mathcal{A}'$ are in the same state. Thus we have transformed $\mathcal{A}$ into $\mathcal{A}'$ with no higher cost while $\pi(\mathcal{A}')$ is strictly lexicographically smaller than $\pi(A)$. Applying such transformations iteratively proves the lemma. $\square$

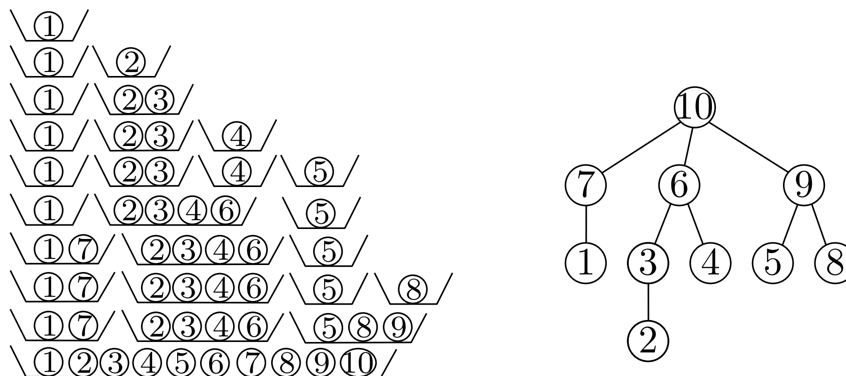

Figure 3: How a ball-shuffling process is modeled as a tree.

The recurrence. In view of Lemma 1, we can model any ball-shuffling process with only merging shuffles as a tree T. An example is shown in Figure 3. More precisely, each ball is modeled as a node in T, and a bin is modeled as a subtree. Then the tree T is inductively defined as follows. If a node (ball) v lands in an empty bin when it arrives, it is a leaf of T; the bin (now containing only v) then corresponds to the singleton subtree v. Otherwise, suppose v is merged with $k \geq 1$ nonempty bins, then v has the k subtrees corresponding these k bins as its subtrees. It is easy to see that the cost of the shuffle for accommodating v is exactly the size of the subtree rooted at v, so we can correspondingly define the cost of T as the sum of the sizes of all its subtrees. Let $v_1, \ldots, v_b$ be the b nodes of T in the arrival order. For each i, let $T^{(i)}$ be the portion of T formed by $\{v_1, \ldots, v_i\}$, which is a forest. Note that the number of trees in $T^{(i)}$ is exactly the number of bins being used after v_i arrives. Thus the problem becomes finding the minimum cost of any T subject to the condition that $T^{(i)}$ has at most t trees for all i. Of course, this tree formulation requires that all balls be merged into one bin in the end, which is not necessarily the case for the ball-shuffling problem, but this only has an additional cost of b.

Let $f_t(b)$ be the minimum cost of any T with b nodes such that $T^{(i)}$ has at most t trees for all i. We will lower bound $f_t(b)$ by induction on t and b. The base cases are $f_t(1) = 1$ for any t, and $f_1(b) = 1 + 2 + \cdots + b = \frac{1}{2}b(b+1)$ for any b. For $t, b \geq 2$, consider the root of T, and suppose it has k subtrees $T_1, \ldots, T_k$, ordered by the arrival times of their earliest members (see e.g. Figure 3). Because $T^{(i)}$ has at most t trees, $T_1^{(i)}$ must also have at most t trees. However, $T_2^{(i)}$ cannot have t trees for any i, because at least one node in T_1 arrives before any node of T_2. If $T_2^{(i)}$ had t trees for some i, then $T_1^{(i)}$ and $T_2^{(i)}$ together would have more than t trees. In general, we can argue that $T_j^{(i)}$ has at most $t - j + 1$ trees for any i. Thus T_j has a minimum cost of $f_{t-j+1}(x_j)$, where x_j is the subtree size of T_j. Considering all possible k and ways to partition $b-1$ nodes into k subtrees, we have

$$f_t(b) \geq \min_{k, x_1 + \cdots + x_k = b-1} f_t(x_1) + f_{t-1}(x_2) + \cdots + f_{t-k+1}(x_k) + b.$$

In fact, we can attain this minimum cost by handling the first x_1 balls using the optimal T_1 with t bins, then the next x_2 balls using the optimal T_2 with $t-1$ bins, and so on so forth. Thus, we actually have a tight recurrence:

$$f_t(b) = \min_{k, x_1 + \cdots + x_k = b-1} f_t(x_1) + f_{t-1}(x_2) + \cdots + f_{t-k+1}(x_k) + b.$$

If we define $f_t(0) = 0$ for all t and allow the x_i's to be 0, we can remove k from consideration:

$$f_t(b) = \min_{x_1 + \cdots + x_t = b-1} f_t(x_1) + f_{t-1}(x_2) + \cdots + f_1(x_t) + b. \tag{3}$$

Rewriting (3) as

$$f_t(b) = \min_{x_1} \left(f_t(x_1) + x_1 + \min_{x_2 + \cdots + x_t = b-1-x_1} (f_{t-1}(x_2) + \cdots + f_1(x_t) + b - x_1) \right),$$

we notice that the second min is exactly $f_{t-1}(b - x_1)$ according to recurrence (3). Thus we obtain a simplified recurrence:

$$f_t(b) = \min_{0 \leq x < b} f_t(x) + f_{t-1}(b-x) + x. \tag{4}$$

The $+x$ term in (4) will create a lot of technical difficulties as one tries to solve the recurrence, so we perform the following maneuver to make it into a $+b$ term. Defining $g_t(b) = f_t(b) + tb$, (4) becomes

$$g_t(b) - tb \quad = \quad \min_{0 \leq x < b} g_t(x) - tx + g_{t-1}(b-x) - (t-1)(b-x) + x$$

$$\Leftrightarrow \qquad g_t(b) \;=\; \min_{0\le x<b} g_t(x) + g_{t-1}(b-x) + b. \tag{5}$$

Below, based on (5) we will first prove that

$$g_t(b) \ge \frac{1}{6} t b^{1+1/t} \tag{6}$$

for all t and b. This almost gives us what we want, but soon one realizes that $b^{1/t}$ becomes a constant that approaches 1 when $t = \Omega(\log b)$, so that $\frac{1}{6} t b^{1+1/t} - tb$ becomes negative. The constant $\frac{1}{6}$ can be pushed to $\frac{1}{2}$ if one is willing to spend more efforts (and by sacrificing the constant in front of $1/t$), but that will not help much. One intuitive explanation is that, although the recurrence (5) simplifies the derivation for $g_t(b)$, it "blurs" the boundary between $f_t(b)$ and tb. Realizing that we just need to prove $f_t(b) = \Omega(b \log b)$ for $t = \Theta(\log b)$, we adopt a more direct approach for these t's. More precisely, in the second phase of the induction, we will prove that $f_t(b) \ge \frac{b \log^2 b}{ct}$ directly using (4) for all $t < \alpha \log b$, where c is a sufficiently large constant. Both phases of the induction will be on (t, b) in lexicographical order.

The induction, phase one. The base cases of (6) are easily established: For $t = 1$, $g_1(b) = f_1(b) + b = \frac{1}{2} b(b+1) + b \ge \frac{1}{6} b^2$; for $b = 1$, $g_t(1) = f_t(1) + t = 1 + t \ge \frac{1}{6} t$. Next, assuming that the lower bound (6) holds on $g_t(x)$ for all $x \le b-1$ and $g_{t-1}(x)$ for all $x \le b$, we will prove the lower bound on $g_t(b)$.

From the recurrence (5) and the induction hypothesis, we have

$$g_t(b) \ge \min_{0\le x<b} \frac{1}{6} t x^{1+1/t} + \frac{1}{6}(t-1)(b-x)^{1+1/(t-1)} + b.$$

So, it suffices to prove

$$\frac{1}{6} t x^{1+1/t} + \frac{1}{6}(t-1)(b-x)^{1+1/(t-1)} + b \ge \frac{1}{6} t b^{1+1/t} \tag{7}$$

for all $0 \le x < b$. Setting $x = \lambda b$ where $0 \le \lambda < 1$, (7) becomes

$$\frac{1}{6} t(\lambda b)^{1+1/t} + \frac{1}{6}(t-1)((1-\lambda)b)^{1+1/(t-1)} + b \;\ge\; \frac{1}{6} t b^{1+1/t}. \tag{8}$$

Dividing by $\frac{1}{6} t b^{1+1/t}$ on both sides, (8) is equivalent to

$$\lambda^{1+\frac{1}{t}} + \frac{t-1}{t}(1-\lambda)^{1+\frac{1}{t-1}} b^{\frac{1}{t(t-1)}} + \frac{6}{t b^{1/t}} \ge 1. \tag{9}$$

Since $\lambda^{1+\frac{1}{t}} \ge \lambda^{1+\frac{1}{t-1}}$, to prove (9), it suffices to prove

$$\lambda^{1+\frac{1}{t-1}} + \frac{t-1}{t} b^{\frac{1}{t(t-1)}} (1-\lambda)^{1+\frac{1}{t-1}} \ge 1 - \frac{6}{t b^{1/t}}. \tag{10}$$

It is easy to see that the derivative of the LHS of (10) is monotonically increasing, so it achieves its only minimum at the point where its derivative is zero, namely when (this is where replacing the $+x$ with $+b$ in (5) greatly simplifies things)

$$\left(1 + \frac{1}{t-1}\right) \lambda^{\frac{1}{t-1}} \;=\; \frac{t-1}{t} b^{\frac{1}{t(t-1)}} \left(1 + \frac{1}{t-1}\right) (1-\lambda)^{\frac{1}{t-1}},$$

$$\text{or} \quad \lambda = \left(\frac{t-1}{t}\right)^{t-1} b^{1/t}(1-\lambda),$$
$$\lambda = \frac{(\frac{t-1}{t})^{t-1} b^{1/t}}{(\frac{t-1}{t})^{t-1} b^{1/t} + 1}.$$

Plugging this λ into the LHS of (10) while letting $\gamma = (\frac{t-1}{t})^{t-1} b^{1/t}$ to simplify notation, we get

$$\frac{\gamma^{1+\frac{1}{t-1}} + \gamma^{\frac{1}{t-1}}}{(\gamma+1)^{1+\frac{1}{t-1}}} = \frac{\gamma^{\frac{1}{t-1}}(\gamma+1)}{(\gamma+1)^{1+\frac{1}{t-1}}} = \left(\frac{\gamma}{\gamma+1}\right)^{\frac{1}{t-1}}.$$

Since $b^{1/t} = (\frac{t}{t-1})^{t-1}\gamma < e\gamma$, the RHS of (10) is at most $1 - \frac{6}{e\gamma t} \leq 1 - \frac{3}{e\gamma(t-1)}$ (since $t \geq 2$). Thus, to have (10), we just need to have

$$\left(\frac{\gamma}{\gamma+1}\right)^{\frac{1}{t-1}} \geq 1 - \frac{3}{e(t-1)\gamma},$$
$$\text{or} \quad \frac{\gamma}{\gamma+1} \geq \left(1 - \frac{3}{e(t-1)\gamma}\right)^{t-1} = \left(1 - \frac{3}{e(t-1)\gamma}\right)^{\frac{e(t-1)\gamma/3}{e\gamma/3}}$$
$$\Leftarrow \quad \frac{\gamma}{\gamma+1} \geq e^{-\frac{3}{e\gamma}}$$
$$\Leftrightarrow \quad 1 + \frac{1}{\gamma} \leq e^{\frac{3}{e\gamma}}$$
$$\Leftarrow \quad 1 + \frac{1}{\gamma} \leq 1 + \frac{3}{e\gamma},$$

which is clearly true.

The induction, phase two. When $t \leq \frac{1}{2}\log b$, $\frac{1}{6}tb^{1+1/t} \geq \frac{1}{6}e^2 tb \approx 1.23tb$, so the result of the first phase already gives us $f_t(b) = g_t(b) - tb = \Omega(tb^{1+1/t})$, as desired. In this phase, we prove that, for some constant c sufficiently large (which will depend on α),

$$f_t(b) \geq \frac{b\log^2 b}{ct} \tag{11}$$

for all $\frac{1}{2}\log b < t < \alpha \log b$, which implies that $f_t(b) = \Omega(b \log b) = \Omega(tb^{1+1/t})$ for these t's. Note that we only need to prove this for any constant α that is sufficiently large.

The base case of (11) includes all t, b such that $t \leq \frac{1}{2}\log b$. From the phase one result, for $t \leq \frac{1}{2}\log b$ we have $f_t(b) = \Omega(tb^{1+1/t})$, which is at least $\frac{b\log^2 b}{ct}$ for some constant c large enough. Indeed, if $t = \log b/\psi(b)$ for any non-decreasing function $\psi(b)$ of b, $tb^{1+1/t} = e^{\psi(b)}/\psi(b) \cdot b\log b$, while $b\log^2 b/t = \psi(b) b \log b$, and it is clear that $e^{\psi(b)}/\psi(b) = \Omega(\psi(b))$.

Now assuming that (11) holds on $f_t(x)$ for all $x \leq b-1$ and $f_{t-1}(x)$ for all $x \leq b$, we will prove the lower bound on $f_t(b)$. From the recurrence (4) and the induction hypothesis, we have

$$f_t(b) \geq \min_{0 \leq x < b} \frac{x\log^2 x}{ct} + \frac{(b-x)\log^2(b-x)}{c(t-1)} + x.$$

Setting $x = \lambda b$ where $0 \leq \lambda < 1$, we just need to prove that

$$\frac{\lambda b \log^2(\lambda b)}{ct} + \frac{(1-\lambda)b\log^2((1-\lambda)b)}{c(t-1)} + \lambda b \geq \frac{b\log^2 b}{ct} \tag{12}$$

for all $0 \le \lambda < 1$. Dividing by $\frac{b}{ct}$ on both sides of (12), it becomes

$$\lambda \log^2(\lambda b) + \left(1 + \frac{1}{t-1}\right)(1-\lambda)\log^2((1-\lambda)b) + \lambda ct \ge \log^2 b. \tag{13}$$

Rewriting (13), we have

$$\begin{aligned}
\lambda\left(-2\log b \log\frac{1}{\lambda} + \log^2\frac{1}{\lambda}\right) + (1-\lambda)\left(-2\log b\log\frac{1}{1-\lambda} + \log^2\frac{1}{1-\lambda}\right) & & \\
+\frac{1}{t-1}(1-\lambda)\left(\log^2 b - 2\log b\log\frac{1}{1-\lambda} + \log^2\frac{1}{1-\lambda}\right) + \lambda ct &\ge& 0 \\
\Leftarrow -2\lambda\log b\log\frac{1}{\lambda} - 2(1-\lambda)\log b\log\frac{1}{1-\lambda} + \frac{1-\lambda}{t-1}\left(\log^2 b - 2\log b\log\frac{1}{1-\lambda}\right) + \lambda ct &\ge& 0. \quad (14)
\end{aligned}$$

Since $\frac{1}{2}\log b < t < \alpha\log b$, to prove (14), it suffices to prove

$$-2\lambda\log b\log\frac{1}{\lambda} - 2(1-\lambda)\log b\log\frac{1}{1-\lambda} + \frac{1-\lambda}{\alpha\log b}\log^2 b - \frac{1-\lambda}{\frac{1}{2}\log b}2\log b\log\frac{1}{1-\lambda} + \frac{c}{2}\lambda\log b \ge 0,$$

which is

$$\frac{1-\lambda}{2\alpha} + \frac{c}{4}\lambda \ge \lambda\log\frac{1}{\lambda} + (1-\lambda)\log\frac{1}{1-\lambda} + \frac{2(1-\lambda)\log\frac{1}{1-\lambda}}{\log b}. \tag{15}$$

Note that $\lambda\log\frac{1}{\lambda} + (1-\lambda)\log\frac{1}{1-\lambda} \le \log 2 \approx 0.69$ (it maximizes at $\lambda = \frac{1}{2}$). Below we show that if we choose $c = 12\alpha^2$, then (15) is true for all λ, provided that α is a sufficiently large constant.

If $\lambda \ge 1/\alpha^2$, then the LHS of (15) is at least $\frac{c}{4}\lambda \ge 3$, while the RHS of (15) is $\le 0.69 + \frac{2\log\frac{1}{1-\lambda}}{\log b} \le 0.69 + \frac{2\log b}{\log b} = 2.69$. If $\lambda \le 1/\alpha^2$, then the LHS of (15) is at least $\frac{1}{2\alpha}$ (minimized at $\lambda = 0$ for $\alpha > 1/6^3$). For the RHS of (15), $\lambda\log\frac{1}{\lambda} + (1-\lambda)\log\frac{1}{1-\lambda}$ is increasing in the region $0 \le \lambda \le 1/\alpha^2$ (for $\alpha \ge 1/\sqrt{2}$), so is at most $\frac{1}{\alpha^2}\log\alpha^2 + \left(1 - \frac{1}{\alpha^2}\right)\log\left(1 + \frac{1}{\alpha^2-1}\right) \le \frac{2}{\alpha^2}\log\alpha + \frac{1}{\alpha^2-1}$, which is smaller than $\frac{1}{4\alpha}$ for sufficiently large α. The term $\frac{2(1-\lambda)\log\frac{1}{1-\lambda}}{\log b}$ is at most $2\log\frac{1}{1-\lambda} \le 2\log\left(1 + \frac{1}{\alpha^2-1}\right) \le \frac{2}{\alpha^2-1}$, which is also smaller than $\frac{1}{4\alpha}$. This completes the entire proof.

4 Final Remarks and Open Problems

We have introduced dynamic indexability and derived lower bounds for dynamic range query indexes in this model. The lower bounds are tight for all practical values of the model parameters, but still there are some intriguing theoretical questions left to be addressed.

We will assume $N > M^2$ to simplify the following discussions. Let us focus on the most interesting tradeoff point $u = O(\frac{1}{B}\log N), q = O(\log N)$. We have proved a matching lower bound for the parameter range $N = B^{O(1)}$. The obvious question is: are the bounds tight for all values of N, ideally not bounded in terms of B? The answer is, unfortunately, no. For example, the exponential tree [3] has $q = O(\sqrt{\log N/\log\log N} + K/B)$ and $u = O(\sqrt{\log N/\log\log N})$. When $B = o(\sqrt{\log N\log\log N})$, this beats the B-tree tradeoff. So the right question to pose should be the following: for what values of B is the B-tree tradeoff optimal? In this paper we get to $B = \Omega(N^\epsilon)$ for any constant $\epsilon > 0$; the exponential tree

implies that this is not true for $B = o(\sqrt{\log N \log\log N})$. Can we narrow this gap? For those small B's where B-trees are not optimal, what is the right query-update tradeoff?

Central to the indexability model is the *indivisibility* assumption, i.e., each data record is manipulated as an atomic element. The same assumption has been made in other external memory lower bounds like sorting and permuting [1]. A major open problem in external memory algorithms is whether this assumption is necessary. There are techniques that do not respect the atomicity of data records (most notably, those based on hashing), but they are not better than the indivisibility-based techniques when B is sufficiently large. The only exception is a very recent surprising result by Iacono and Pătraşcu [16], where they give a hashing-based solution for the *dictionary* problem that is better than the dynamic B-trees for (essentially) all values of B. Their data structure, however, does not work for range reporting. It remains an intriguing question whether one can break the indivisibility barrier for range reporting or prove a lower bound without assuming indivisibility (say, in the cell-probe model).

Acknowledgment. The author would like to thank an anonymous reviewer for suggesting ways to simplify the proof of Theorem 2.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] S. Alstrup, G. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proc. ACM Symposium on Theory of Computing*, pages 476–482, 2001.

[3] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 53(3), Article 13, 2007.

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

[5] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symposium on Principles of Database Systems*, pages 346–357, 1999.

[6] L. Arge, V. Samoladas, and K. Yi. Optimal external memory planar point enclosure. *Algorithmica*, 54(3):337–352, 2009.

[7] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[8] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

[9] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, 2007.

[10] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1:301–358, 1980.

[11] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554, 2003.

[12] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[13] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[14] J. M. Hellerstein, E. Koutsoupias, D. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM*, 49(1):35–55, 2002.

[15] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proc. ACM Symposium on Principles of Database Systems*, pages 249–256, 1997.

[16] J. Iacono and M. Pătraşcu. Using hashing to solve the dictionary problem (in external memory), 2011. http://arxiv.org/abs/1104.2799.

[17] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *Proc. International Conference on Very Large Data Bases*, pages 16–25, 1997.

[18] C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data waresshouse insertion. In *Proc. International Conference on Very Large Data Bases*, pages 235–246, 1999.

[19] E. Koutsoupias and D. S. Taylor. Tight bounds for 2-dimensional indexing schemes. In *Proc. ACM Symposium on Principles of Database Systems*, pages 52–58, 1998.

[20] C. W. Mortensen, R. Pagh, and M. Pătraşcu. On dynamic range reporting in one dimension. In *Proc. ACM Symposium on Theory of Computing*, pages 104–111, 2005.

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

[22] V. Samoladas and D. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proc. ACM Symposium on Principles of Database Systems*, pages 44–51, 1998.

[23] A. C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.