Storing Matrices on Disk: Theory and Practice Revisited

Yi Zhang
Duke University
yizhang@cs.duke.edu

Kamesh Munagala
Duke University
kamesh@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

We consider the problem of storing arrays on disk to support scalable data analysis involving linear algebra. We propose ***Linearized Array B-tree***, or *LAB-tree*, which supports flexible array layouts and automatically adapts to varying sparsity across parts of an array and over time. We reexamine the B-tree splitting strategy for handling insertions and the flushing policy for batching updates, and show that common practices may in fact be suboptimal. Through theoretical and empirical studies, we propose alternatives with good theoretical guarantees and/or practical performance.

1 Introduction

Arrays are one of the fundamental data types. Vectors and matrices, in particular, are the most natural representation of data for many statistical analysis and machine learning tasks. As we apply increasingly sophisticated analysis to bigger and bigger datasets, efficient handling of large arrays is rapidly gaining importance. In the *RIOT* project [17], we are building a system to support scalable statistical analysis of massive data in a "transparent" fashion, which allows users to enjoy the convenience of languages like R and MATLAB with built-in support for vectors/matrices and linear algebra, without rewriting code to use systems like databases that scale better over massive data.

Scalability requires efficient handling of disk-resident arrays. Our target applications make prevalent use of high-level, whole-array operators such as matrix multiply, inverse, and factorization, but low-level, element-wise reads and writes are also possible. We have identified the following requirements for an array storage engine (more detailed motivating examples can be found in the technical report version of this paper [18]):

1. We must support different array access patterns (including those that appear random). Our storage engine should allow a user or optimizer to select from a variety of storage layouts, as many whole-array operators have access patterns that prefer specific storage layouts: e.g., I/O-efficient matrix multiply prefers row, column, or blocked layouts, while FFT the bit-reversal order. Moreover, a single array may be used in operators with different access patterns; instead of converting the storage layout for every use, sometimes it is cheaper to allow access patterns that do not match the storage layout, even if it makes accesses random. Finally, some operators' access patterns inherently contain some degree of randomness that cannot be removed by storage layouts, e.g., LU factorization with partial pivoting.
2. We must handle updates. One common update pattern is populating an array one element at a time in some order, which may or may not be the same as the storage layout order. Handling updates goes beyond bulk loading: some operators, such as LU factorization, iteratively update an array and read previously updated values, which means that we cannot simply log all updates without efficiently supporting interleaving (and sometimes random) reads to updated values.
3. We want the storage format to automatically adapt to array sparsity. For a sparse array, we want to avoid wasting space for elements that are zero (or some other default value), which can be done by storing array indices and values only for non-zero elements. On the other hand, for a dense array, we want to avoid the overhead of storing array indices by densely packing the values and inferring their indices from storage positions. In practice, there is no obvious delineation between "sparse" and "dense"; sparsity often varies across parts of an array and over time, and is difficult to predict in advance. For example, consider an application program that updates an initially empty (all-zero) matrix one element at a time in random order according to some ongoing computation. The matrix may turn out dense, sparse, or partly dense (e.g., mostly upper-triangular); regardless of its final content, our storage engine should store the matrix in a way that provides good performance throughout the update sequence, without user intervention.

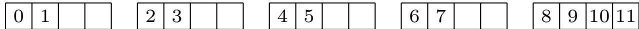
There has been a myriad of approaches to storing arrays on disk, but many fail to meet all requirements above. Targeting in-memory computation, popular platforms for statistical computing such as R and MATLAB offer separate dense and sparse storage formats, but these formats do not adapt to varying sparsity across parts of an array and over time, and users must choose one format in advance. *Compressed sparse column*, used by MATLAB and representative of popular sparse formats, does not support updates or random accesses for disk-resident arrays.[1] Alternatively, a database system can store an array as a table with columns representing array index and value, but the overhead is high for dense arrays. It is generally believed that special support for arrays is needed in database systems, either through user-defined extensions or by completely new designs [1, 13, 6, 15]. Section 2 surveys additional related work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 37th International Conference on Very Large Data Bases, August 29th - September 3rd 2011, Seattle, Washington.
Proceedings of the VLDB Endowment, Vol. 4, No. 11
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

[1] For memory-resident arrays, this format is easier to search but still inefficient to update.

A promising approach is to leverage B-tree [3]. To handle multi-dimensional arrays, we use *linearization*, which maps a multi-dimensional coordinate to a 1-d array index according to a *linearization function* that offers control over data layout. To adapt to varying sparsity, we apply the idea of compression, allowing each B-tree leaf to switch dynamically between sparse and dense formats according to the array density within the leaf. Simply outfitting B-tree with these features, however, falls short of offering optimal performance for arrays, as illustrated below.

Example 1. *Consider sequentially inserting elements of array into an empty B-tree, which is a very common update pattern. Suppose the array has size* 12 *and a B-tree leaf can hold at most* 4 *records. When a leaf overflows, the standard strategy is split-in-middle, which divides the leaf into two with equal number of records (or as closely as possible). The leaf level of the B-tree after the insertion sequence looks as follows (only record keys are shown). About half of the space is empty, which is particularly wasteful as no future insertions can possibly fill it. The suboptimal space utilization also hurts access performance; e.g., array scans become twice as costly.*



While one can handle sequential insertions as a special case, other patterns that lead to waste are difficult to detect. Are there alternative splitting strategies that are provably resilient against such waste, without knowing the insertion sequence in advance?

Example 2. *A popular trick to speed B-tree updates is to batch them by keeping individual record updates in a memory buffer. When the buffer fills up, we flush the buffered updates by applying them in key order. This approach reduces I/Os by applying multiple updates to the same B-tree leaf with a single leaf access. A large buffer also helps make the leaf accesses more sequential. However, for the following update sequence, the conventional policy of flushing all buffered updates when the buffer is full is not optimal. Here,* K *denotes the number of updates that the buffer can hold, and each* P_i *represents an update of some record on leaf* P_i*.*

$$\underbrace{P_1, \ldots, P_1}_{K/2}, \underbrace{P_2, \ldots, P_2}_{K}, \underbrace{P_3, \ldots, P_3}_{K}, \underbrace{P_4, \ldots, P_4}_{K}, \ldots$$

The flush-all policy incurs two leaf writes (of P_i *and* P_{i+1}*) for every* K *updates. However, the optimal policy would flush all* P_1 *updates after the* $(K/2)$*-th update; subsequently, only one leaf write would be incurred for every* K *updates.*

For this simple insertion sequence, flush-all is only a factor of 2 *worse than the optimal. As we will see later, however, there exist sequences for which flush-all is a factor of* $\Omega(\sqrt{K})$ *worse. Are there flushing policies that offer better competitive ratio in theory or perform better in practice?*

In this paper, we present *LAB-tree* (***L****inearized* ***A****rray* ***B-tree***), the backbone of the RIOT array storage engine, which meets all requirements identified earlier. LAB-tree offers flexible layouts via linearization; it inherits from B-tree efficient support for accesses and updates; and it adapts to varying sparsity by switching between dense and sparse storage formats automatically on a per-leaf basis. LAB-tree reexamines the leaf splitting strategies and batched update flushing policies, for which common practices have been rarely questioned. We present theoretical and empirical results that contribute to the fundamental understanding of these problems.

These results challenge the common practices. For leaf splitting, exploiting the fact that the domain of array indices is bounded and discrete, we devise a strategy that naturally produces trees with "no-dead-space," often twice as efficient as those produced by split-in-middle. This advantage does incur a fundamental trade-off—in the worst-case, split-in-middle has competitive ratio 2, while this strategy has 3, which is the best possible for any "no-dead-space" strategy. Nonetheless, on common workloads, this strategy consistently and significantly outperforms split-in-middle.

For update batching, we give a flushing policy with competitive ratio $O(\log^3 K)$ in the worst case, beating flush-all's $\Omega(\sqrt{K})$. For common workloads, however, flush-all actually performs better in practice. On the other hand, starting from a simple policy with a poor competitive ratio of $\Omega(K)$, we devise a randomized variant that incurs fewer number of I/Os than flush-all for some workloads (and comparable numbers for others). Our approach can be seen as bringing to the update batching problem the same level of rigor as in the study of caching (though caching results do not carry over).

Finally, we note that our techniques are easy to implement as they do not require intrusive modifications to the conventional B-tree. Also, many of our results generalize to other settings: the idea of "no-dead-space" splitting makes sense for other discrete, ordered key domains; theoretical analysis of update batching generalizes to other block-oriented or distributed data structures.

2 Related Work

Database systems have been extended with support for arrays, and more specifically, linear algebra. Besides storing arrays as tables whose rows correspond to individual array elements, UDTs and UDFs are popular implementation options (e.g., [14, 6]). In general, these approaches can be seen as dividing an array into chunks and storing each chunk in a database row as a unit of access. SQL can express many linear algebra operations by calling UDFs that operate on chunks or pairs of chunks. Database indexing is used for accessing chunks. While this paper does not store arrays in databases, many ideas, such as linearization, dynamic storage format, and update batching, are readily applicable by regarding a table of chunks as a block-oriented storage structure.

There has also been work building database systems specializing in arrays (e.g., *RasDaMan* [1], *ArrayDB* [13], and *SciDB* [15]). These approaches divide arrays into rectangular chunks, and often rely on spatial indexing to retrieve chunks in high-dimensional arrays. Our approach of linearization supports more layouts (e.g., bit-reversal) and avoids the difficulty of high-dimensional indexing. One reason for this different approach is that we focus less on ad hoc region-based retrieval, but more on whole-matrix operations with more predictable but specific access patterns. Nonetheless, it would be interesting to see how our ideas can be applied in their settings (e.g., linearization, alternative index reorganization and update buffering methods) and vice versa (e.g., allowing replication of boundary elements between neighboring chunks as in SciDB).

Linearization is frequently used for multi-dimensional indexing. *UB-tree* [2] is the most related to our work in this regard. While UB-tree linearizes arrays using Z-order, LAB-tree provides more linearization options to match different application needs (with a similar goal as *RodentStore* [7], but at a different level). More importantly, we reexamine index reorganization and update buffering practices, which UB-tree does not address.

There is no shortage of B-tree tricks [3, 12] aimed at improving its efficiency. Prefix B-tree compression, for example, is a more general form of compression than our dynamic leaf format, though its generality also carries some overhead. There is also work on alternative splitting strategies, such as avoid splitting by scanning adjacent nodes for free space [9]. Most of these techniques are orthogonal to ours and may further improve LAB-tree in some cases. We are not aware of any previous work on alternative splitting strategies for bounded, discrete key domains and how they interact with compression. Work on update batching dates back to Lohman

et al. [11]. Like us, instead of a complete reorganization, Lang et al. [10] propose accumulating insertions in a batch, sorting them by key, and applying them to B-tree by traversing from left to right and backtracking along root-to-leaf paths when necessary. Our contribution to the update batching problem lies in analyzing and questioning the standard practice of flushing all buffered updates.

3 Overview of LAB-Tree

Based on B-tree, LAB-tree introduces modifications and extensions designed for arrays: linearization (this section), new leaf splitting strategies (Section 4.1), dynamic leaf storage format (Section 4.2), and alternative flushing policies for update batching (Section 5).

Each LAB-tree has a *linearization function* that specifies the storage layout of the array. For an array of dimension d and size $N_1 \times \cdots \times N_d$, a linearization function $f : [0, N_1) \times \cdots \times [0, N_d) \rightarrow [0, N_1 \times \cdots \times N_d)$, where all intervals are over $\mathbb{N}_0$, is a bijection that maps each d-d array index to a 1-d array index. When $d = 1$, f is a permutation. Conceptually, LAB-tree indexes the values of array elements by their linearized array indices; i.e., the element of array A with index $\vec{\imath} = \langle i_0, \ldots, i_d \rangle$ is indexed as the key-value pair $(f(\vec{\imath}), A[\vec{\imath}])$. Popular layouts, such as row-major, column-major, blocked, Z-order, bit-reversal, can be easily and succinctly defined as linearization functions (see [18] for examples). LAB-tree supports arbitrary user-defined linearization functions; for convenience and efficiency, however, frequently used ones have support built in.

Each LAB-tree also has a *default value* (often 0) for array elements. Conceptually, LAB-tree only indexes elements whose values differ from the default. A new, "empty" array is filled with the default value. Setting a default-valued element to non-default value amounts to an *insertion*; the inverse operation amounts to a *deletion*. For convenience and without loss of generality, we will assume the default value to be 0 for the remainder of the paper.

With LAB-tree, we support three types of array accesses: **1)** *Accessing an element by its array index* $\vec{\imath}$, which amounts to accessing the LAB-tree with key $f(\vec{\imath})$. **2)** *Accessing elements of an array via an iterator with linearization function* g, which specifies the access order and may differ from the linearization function f for controlling the storage order. The i-th element in the access order has LAB-tree key $f(g^{-1}(i))$. We implement optimizations to speed up key calculation (see [18]), including incremental computation of $f \circ g^{-1}$ and detecting the special (but common) case of $f = g$. We also support an option to iterate over only non-zero elements. **3)** *Reading/writing elements in a specified hyper-rectangle in the array index space*. This type of access is common in I/O-efficient matrix algorithms (such as multiply) that process matrices a chunk at a time, whose size depends on the amount of available memory. Supporting such accesses as batch operations allows us to avoid the overhead of iterator calls and provide more efficient implementation for built-in linearization functions.

4 Efficiency Through Better Space Utilization

This section tackles B-trees' efficiency problem from two angles: splitting strategy (Section 4.1) and leaf storage format (Section 4.2). Both aim at improving space utilization, which, as validated by our empirical study (Section 4.3), is largely in line with the goal of improving time efficiency as well. We show that by exploiting the special characteristics of arrays, LAB-trees can achieve much better performance than conventional B-trees.

4.1 Splitting Strategy Revisited

As motivated in Section 1, the standard B-tree splitting strategy can lead to lots of wasted space within leaves that will never get used. In the following, we formalize the desirable properties of a splitting strategy, propose several alternatives, and discuss their properties.

We begin with some terminology. Let κ denote the *leaf capacity*, or the maximum number of records that can be stored in a leaf of the index. Each leaf has a *(key) range*, which contains all keys of records stored in this leaf. The set of all leaf ranges forms a disjoint partitioning of the key domain. Since our index stores a 1-d array, a leaf range is an interval $[l, u)$, where l and u are the lower bound (inclusive) and upper bound (exclusive) of the 0-based array indices stored in the leaf. We define the *density* of a leaf ℓ, denoted $\rho(\ell)$, as the number of records in ℓ divided by its capacity. Density can be similarly defined for a set of leaves or the entire index.

When a record needs to be inserted into a leaf with range $[l, u)$ and already κ records (thereby causing it to overflow), a *splitting strategy* chooses a *splitting point* x, such that the original leaf is split into two leaves with ranges $[l, x)$ and $[x, u)$. A splitting strategy operates in an *online* fashion; i.e., it processes the current insertion without knowledge of future insertions. To ensure low runtime overhead, we consider only *local* splitting strategies, i.e., ones that do not read or modify leaves other than the one being inserted into. Also, we focus on *leaf* splitting strategies; splitting at upper levels of the index has little impact on the overall space and efficiency, and we simply follow the standard B-tree strategy.

The standard B-tree leaf splitting strategy is as follows:

- **Split-in-Middle.** Given an overflowing leaf with $\kappa + 1$ records with keys $i_0, i_1, \ldots, i_\kappa$, this strategy chooses the splitting point to be $x = i_j$, where $j = \lfloor (\kappa + 1)/2 \rfloor$.

There are two desirable properties that a good splitting strategy should have: *bounded space consumption* and *no dead space*. The space consumption of a splitting strategy can be measured by its competitive ratio with respect to an optimal offline algorithm. Formally, a splitting strategy Σ is α*-competitive* if, for any insertion sequence S, the number of leaves produced by Σ at the end of S is less than α times that produced by an optimal offline algorithm, within an additive constant. Knowing the entire S, the optimal offline algorithm basically stores all non-zero array elements compactly, so an array with range $[0, N)$ and $n \leq N$ non-zero elements can be stored in $\lceil n/\kappa \rceil$ leaves.[2]

Split-in-middle is clearly 2-competitive, because it always generates leaves that are half full. It turns out that this competitive ratio is the best we can hope for: we show that no deterministic local splitting strategy can have a competitive ratio of less than 2 (Theorem 1 in appendix).

A second desirable property of splitting strategies is *no-dead-space*. By "dead space" we mean empty slots in leaves that can never be filled by future insertions. For example, every leaf except the last one in Example 1 has two slots of dead space. Note that the notion of dead space is special to unique indexes with discrete key domains. General B-tree leaves do not have dead space; it is always possible to insert a record with a duplicate key, or a record between two adjacent existing keys (up to some limit—precision of floating-point keys or maximum length of string keys). Formally, we define the no-dead-space property as follows. Without loss of generality, assume that the array size is a multiple of κ.[3]

Definition 1 (No-Dead-Space)**.** *A splitting strategy Σ is* no-dead-space *if for any index state Σ may result in, there exists a future insertion sequence that causes all leaves to be full under Σ.*

[2] We assume standard B-tree leaf format for now; optimizations for dense array regions are discussed later in Section 4.2.

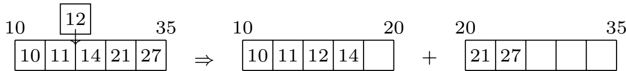
[3] Otherwise, for an array with range $[0, N)$, the last leaf can have $N - \kappa\lfloor N/\kappa \rfloor$ slots of dead space.

As we have seen Section 1, split-in-middle does not have this property. But how important is no-dead-space, given that split-in-middle already has the best possible competitive ratio? Consider any array (or a region within an array) with density ϱ. A strategy that is no-dead-space would be guaranteed to have a competitive ratio of no more than $1/\varrho$ for storing the array (or the dense region). In contrast, regardless of density, split-in-middle may well take twice the minimum space required, as illustrated in Example 1. Thus, split-in-middle is less attractive than a no-dead-space strategy when $\varrho > 1/2$, which is a rather common case in our setting. For example, all dense matrices fall into this case, unless they are at the early stage of being populated in non-sequential order. Hence, no-dead-space is an important property that focuses less on the worst case and more on the common case of dense matrices or dense regions in matrices.

We propose a novel strategy that is naturally no-dead-space:

- **Split-Aligned.** Given an overflowing leaf ℓ with range $[l, u)$, this strategy chooses the splitting point x to be a multiple of κ that minimizes the difference between the number of records in $[l, x)$ and that in $[x, u)$. If multiple values of x satisfy the condition, the one that minimizes $\left|x - \frac{l+u}{2}\right|$ is chosen.

In other words, split-aligned favors a split that is most balanced, like split-in-middle, but under the condition that the splitting point aligns with $0, \kappa, 2\kappa, \ldots$, i.e., endpoints of the leaf ranges had we laid out all array elements (zero or non-zero) compactly. For example, with $\kappa = 5$, split-aligned will choose the following split:



It is easy to see that, starting with a single leaf with range $[0, N)$, split-aligned is no-dead-space.

An obvious question is how split-aligned does on competitive ratio. Unfortunately, there is a fundamental trade-off between no-dead-space and bounded space consumption—we show that any no-dead-space splitting strategy must have a competitive ratio of at least 3 (Theorem 2 in appendix), which is worse than split-in-middle in the worst case. We also show that split-aligned indeed has a competitive ratio of 3; i.e., it is the best no-dead-space strategy possible (Theorem 3 in appendix). This bound is non-trivial, considering that split-aligned may generate near-empty leaves.

Besides split-in-middle and split-aligned, we also consider:

- **Split-off-Dense.** Given a leaf to split with range $[l, u)$, this strategy first considers two candidate splitting points $l + \kappa$ and $u - \kappa$, which would result in a leaf with range $[l, l + \kappa)$ or one with range $[u - \kappa, u)$, respectively. Note these leaves will never be split further. If either leaf has density greater than 0.5, we choose the splitting point that would result in the leaf with the higher density. Otherwise, we fall back to split-in-middle. Intuitively, this strategy can be seen as a tweak to split-in-middle that first tries to split off a dense leaf that will not split again in the future. It is not hard to see that split-off-dense is no worse than split-in-middle in terms of competitive ratio, but split-off-dense may sometimes do better, e.g., the sequential insertion sequence in Example 1.
- **Split-Defer-Next.** This strategy tries to choose a splitting point that delays the split of either result leaf as much as possible. Suppose we split a leaf ℓ with range $[l, u)$ and keys $i_0, \ldots, i_\kappa$ into leaves ℓ_1 and ℓ_2 with splitting point x. Assuming that each future insertion hits each missing key with equal probability, we can calculate $\tau(x)$, the expected number of future insertions into $[l, u)$ that will cause the first split of either ℓ_1 or ℓ_2, using a formula involving l, u, and $i_0, \ldots, i_\kappa$ (see Remark B.1 in appendix for the formula and its derivation). Split-defer-next choose the splitting point to be $\arg\max_x \tau(x)$. Unfortunately, the formula for $\tau(x)$ is quite involved, and we have no closed-form solution for this maximization problem; therefore, we resort to trying every $x \in \{i_1, \ldots, i_\kappa\}$ in a brute-force fashion.
- **Split-Balanced-Ratio.** This strategy shares the same goal as split-defer-next, but uses a simpler optimization objective that is computationally easier. Given a leaf ℓ, consider the ratio $\chi(\ell)$ between the number of free storage slots in ℓ and the number of keys missing from (and hence can be later inserted into) ℓ's range. Intuitively, a bigger $\chi(\ell)$ means ℓ is less likely to split in the future. Split-balanced-ratio picks the splitting point that maximizes the minimum of the two resulting leaves' ratios (see Remark B.2 for the formula).

Section 4.3 compares these strategies with split-in-middle and split-aligned using common workloads for matrices.

We have only discussed insertions so far. Deletions can be handled using standard B-tree techniques; see Remark B.3. They are not the focus of this paper because we find deletions to be rare in our workloads and hence less important to overall performance.

4.2 Dynamic Leaf Storage Format

As discussed in Section 1, plain B-trees are not efficient for dense arrays. We want LAB-tree to be efficient for dense arrays as well as arrays whose sparsity varies over time and across different regions inside them. To this end, LAB-tree supports two leaf storage formats, *sparse* and *dense*. Different leaves can have different storage formats, and each leaf can switch between the two formats dynamically. A *sparse*-format leaf stores each non-zero array element in its range as a key-value pair; zeros are not stored. Let κ_s denote the *sparse leaf capacity*, i.e., the maximum number of records that can be stored by a sparse-format leaf. A *dense*-format leaf, on the other hand, stores all values (zero or non-zero) of array elements from a continuous subrange of its key range. The key that starts the subrange is also stored, but the other keys in the subrange are not, because they can be simply inferred from the starting key and the entry positions. Let κ_d denote the *dense leaf capacity*, i.e., the maximum length of the subrange, or the maximum number of records that can be stored by a dense-format leaf. Clearly, $\kappa_d > \kappa_s$. For example, if the keys are 64-bit integers and values are 64-bit doubles, then $\kappa_d \approx 2\kappa_s$. This two-format approach can be regarded as a simple compression method, which we feel provides a good trade-off between storage space and access time. More sophisticated compression methods are certainly possible, but they will likely add non-trivial decompression overhead to data accesses.

LAB-tree automatically switches between the two formats when a leaf is written. We call the *effective range* of a leaf ℓ to be the tightest interval containing all keys stored in ℓ. The effective range of ℓ is always contained in the range of ℓ. If an insertion overflows a sparse-format leaf ℓ, and the length of ℓ's effective range (containing all $\kappa_s + 1$ keys) is no greater than κ_d, then we switch ℓ to the dense format without splitting ℓ. Conversely, if an insertion into a dense-format leaf ℓ expands the length of its effective range to greater than κ_d but the total number of records is still below κ_s, then we switch ℓ to the sparse format without splitting ℓ.

The splitting strategies in Section 4.1 need to be modified to work with the dynamic leaf format; for details see Remark B.4.

4.3 Experimental Evaluation

Splitting Strategies on Common Insertion Patterns We first compare the performance of various splitting strategies, for now assuming sparse formats across all leaves. We consider the following patterns for populating an initially empty matrix with row-major

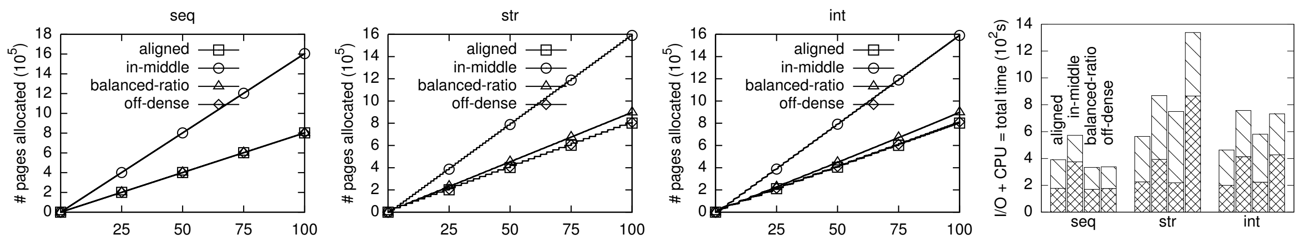


Figure 1: Splitting strategies, with all leaves using the sparse format. In the first three graphs (for *seq*, *str*, and *int*), horizontal axes show the percentage of elements inserted so far; each plot contains one data point every 1000 insertions, and shows one tick every 10^8 insertions. In the last figure, the vertical axis shows the break-down of running time into I/O and CPU, with CPU on top.[4]

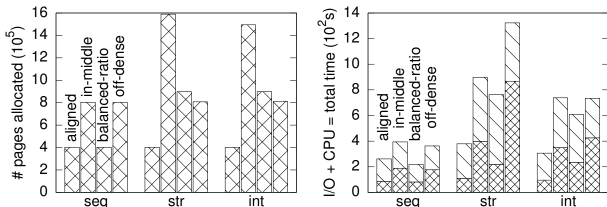


Figure 2: Splitting strategies, with dynamic leaf storage format.

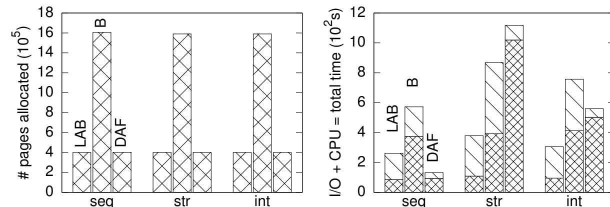


Figure 3: LAB-tree, B-tree, DAF; 20000×20000 **dense matrix.**

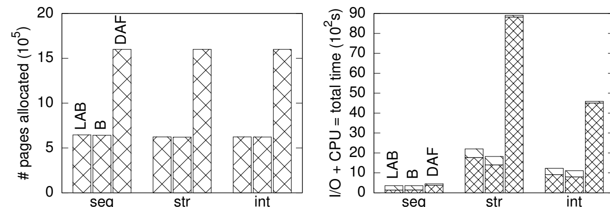


Figure 4: LAB-tree, B-tree, DAF: 40000×40000 **sparse matrix.**

layout: ***seq****(uential)* inserts elements in row-major order; ***str****(ided)* inserts elements in column-major order; ***int****(erleaved)* inserts elements in row- and column-major orders in an interleaving fashion (as in LU factorization); and ***ran****(dom)* inserts elements in random order.

Figure 1 summarizes the results for a 20000×20000 matrix and a 320MB buffer pool; see Remark B.5 for detailed experimental setup. Results on other scales are similar. For this experiment, *ran* is too expensive to run to completion; it takes an hour just to process 4% of the insertions. As its performance is clearly unacceptable regardless of the choice of splitting strategy, we do not discuss *ran* further here. We will, however, revisit *ran* in Section 5.3 because update batching helps improve its performance.

From the first three graphs in Figure 1, we see that standard split-in-middle uses about twice as much space as others throughout the course of each workload. From the last graph, we see that split-in-middle's simpler splitting logic is not enough to make up for its loss in I/O efficiency.[4] On the other hand, split-aligned maintains a noticeable lead ahead split-in-middle in running time, and is the best strategy overall in both space and time efficiency.

As for other strategies, split-off-dense has curiously high running time for *str* despite its low number of I/Os (whose plots are not shown here but are consistent with the first three graphs); a closer examination of the traces reveals that split-off-dense's tendency to generate far more unbalanced leaves than others leads to very scattered I/Os. Split-balanced-ratio has no better space utilization than split-aligned but carries higher CPU overhead. We omit split-defer-next here and subsequently, because it has prohibitive CPU overhead but offers no significant space savings.

Next, we repeat the experiments with dynamic leaf storage format, to study how this feature further affects performance. Figure 2 summarizes the results. All strategies benefit from this feature, but split-aligned benefits more, thanks to its ability to produce leaves that are better aligned (and hence better "prepared") for the dense format. For the more interesting patterns of *str* and *int*, its advantage over split-in-middle widens to a factor of more than 3.5 in terms of space, and more than 1.7 in terms of time; its advantage over other strategies are also more pronounced than in Figure 1. Moreover, the relative performance differences stay the same over the course of the workloads (plots are omitted here, but exhibit the same linear trends as the first three graphs in Figure 1). In conclusion, split-aligned is a clear winner.

Finally, note that these experiments only report the running time of populating the matrix. Split-aligned, with its highest space efficiency, becomes even more appealing if we consider the cost of accessing the matrix subsequently. For other strategies, one could bulk load (and compact) the array at end of the insertion sequence to make subsequent scans more efficient, but doing so would further add to the running time and, for a dense matrix, result in a final tree no better than split-aligned.

LAB-Tree, B-Tree, and Directly Addressable File We now step up a level and compare the performance of LAB-tree (with split-aligned and dynamic leaf storage format), standard B-tree (with split-in-middle and sparse leaf format), and *directly addressable file* (*DAF*). DAF stores all array values compactly in a file, enabling direct lookups and eliminating the need to store array indices or to use extra indirections for indexing. File system optimizations allow us to allocate disk pages for DAF lazily: if a page has never been written (because it contains all zeros), it is never allocated.

First, we repeat the same experiments for a 20000×20000 matrix in Figure 2, and summarize the results in Figure 3. In terms of space utilization, LAB-tree is on par with DAF, the best possible in this case; B-tree is four times worse, because it lacks the dense format and its leaves are mostly half-full. As for running time, the break-down into CPU and I/O offers interesting insights. In terms of CPU time, DAF is the fastest, and B-tree is the slowest; the reasons are that DAF's direct address calculation is simpler than tree lookups, and that searching with the sparse leaf format (which B-tree uses exclusively) is more expensive than the dense format. In

[4]Note that our CPU time accounting includes time spent outside system calls on behalf of I/Os. In particular, time spent on I/Os served from our buffer pool without hitting the disk is counted towards the CPU time instead of the I/O time. In this figure, the CPU time's significant proportion is in part explained by the effectiveness of our buffer pool for these workloads.

terms of I/O time, B-tree suffers from a larger number of I/Os. Surprisingly, DAF has the worst I/O time for *str* and *int*, even though it incurs a similar number of I/Os (not plotted here) as LAB-tree. A closer look shows that DAF generates very scattered I/Os because column-major insertions hit faraway portions of the file. In this regard, LAB- and B-trees are better at placing and moving array elements during the course of these workloads. This observation offers the insight that it can be suboptimal to simply place each element where it should be at the end of the insertion sequence, as the intermediate states of the data structure also affect performance.

In the second set of experiments, we populate a sparse 40000×40000 matrix with 10% randomly distributed non-zero elements. Figure 4 summarizes the results. As expected, DAF really suffers while B-tree shines, as there are not even locally dense regions in this matrix. Despite being unable to exploit any density, LAB-tree maintains comparable performance to B-tree, except that LAB-tree has slightly higher I/O time due to slightly more random I/Os.

From the above two sets of experiments, which straddle the opposite ends of the dense-sparse spectrum, we see that LAB-tree is able to automatically achieve optimal (or close to optimal) performance without manual tuning.

More Experiments on LAB/B/DAF We have conducted more experiments with different matrix sizes and different buffer pool sizes. We have also tested other insertion patterns resulted from different combinations of storage layouts and access patterns (e.g., bit-reversal). All results lead to similar conclusions as the experiments above. Details can be found in [18].

BLAS on UFSparse Stepping up yet another level, we examine how LAB-tree compares with B-tree and DAF for linear algebra operations involving real-world matrices. For the operation, we test matrix multiply, an essential and often performance-critical building block of more sophisticated analysis. We use an I/O-efficient version of the block matrix multiply algorithm, which computes the result matrix one block (submatrix) at a time by reading and multiplying pairs of blocks from the input matrices and accumulating the multiplication results in memory. For multiplying submatrices in memory, we use the BLAS routine `dgemm` if both submatrices have density greater than 0.5, or CHOLMOD [5] (an efficient sparse matrix package) routines otherwise.

For input, we use matrices from UFSparse, the University of Florida Sparse Matrix Collection [8]. To test each storage method, we prepare the input matrices with this method using a *blocked linearization* perfectly matching the pattern of blocks accessed by the I/O-efficient matrix multiply. We multiply each input matrix with itself, and save the result using the same storage method as the input. Here, we discuss results for two matrices, `human_gene2` and `TSOPF_RS_b2383`; more results are available in [18]. We report the total running time, which excludes input preparation but includes writing the result.

For `human_gene2` (14340×14340 and density 8.79%), we use 1500×1500 blocks, and the total running time is 386sec for LAB-tree, 461sec for B-tree, and 1170sec for DAF. DAF suffers from a bloated input file. LAB- and B-trees both perform well, with LAB-tree leading by about 16%. Their input trees are comparable in size, because `human_gene2` looks uniformly sparse. The result matrix turns out fairly dense, so the LAB-tree result is more compact.

For `TSOPF_RS_b2383` (38120×38120 and density 1.11%), we use 4000×4000 blocks, and the total running time is 388sec for LAB-tree, 615sec for B-tree, and 1088sec for DAF. Unlike `human_gene2`, this matrix has a dense region despite its overall sparsity. LAB-tree is able to exploit this local density to widen its lead over B-tree to a factor of 1.6. Its lead over DAF narrows slightly, but is still more than a factor of 2.8.

5 Update Batching

We now turn to the problem of batching index updates in a memory buffer[5] to consolidate writes to disk. To support index access while updates are being buffered, we organize this buffer as an index over the buffered updates; a record lookup would be first checked against this in-memory index. Whenever the buffer is full, we need to flush updates, i.e., applying them in a batch to the underlying disk-resident indexes. As discussed in Section 1, we question the common practice of flushing *all* buffered updates whenever the buffer is full. Section 5.1 presents alternative policies and a theoretical analysis of their performance. Section 5.2 discusses implementation issues and Section 5.3 presents an empirical evaluation.

5.1 Flushing Policies and Analysis

To simplify theoretical analysis, we make some assumptions. First, we view each update to a record r as a request for the disk page (leaf) that contains r or will contain r, and we assume that we know the identities of all requested pages before each flushing action (see Section 5.2 for implementation details). Second, we assume that each flush incurs a fixed cost per update plus a fixed cost per page; multiple updates requesting the same page incur the per-page cost only once for the flush, reflecting the benefit of batching. Because the sum of per-update costs in the end remain the same no matter how we flush, we focus on minimizing the sum of per-page costs over time. Note that this analytical model is an imperfect simplification of reality. For example, it ignores the cost of obtaining page identities (Section 5.2) and that of splitting (which depends on factors such as the splitting strategy). Nonetheless, it provides a reasonable estimate of the true cost, and makes our analysis more generalizable to other batch processing settings.

With these assumptions, we now formally define the problem.

Definition 2. *There are a set of pages $\mathcal{P}$ on disk, and a buffer of capacity K in memory for buffering requests. Every request refers to a page and takes unit space in the buffer.*[6] *A* flushing policy *selects subsets of requests to flush as needed to keep the buffer size capped at K at all times. Flushing requests for the same page incurs unit cost. We are interested in an online flushing policy that minimizes the total cost over a request sequence.*

For brevity, by "buffered" requests we mean all requests eligible for flushing, which include the incoming request. Without loss of generality, we assume a policy only flushes when the buffer is full (any policy can be modified to do so without affecting the cost). We can also assume that if a policy flushes any request for P, it flushes all buffered requests for P; in this case, we simply say it flushes P.

As it may have occurred to the reader, this problem looks similar to cache replacement [16]. Unfortunately, known results on caching do not carry over. Although caching has been generalized to cases where pages can have varying sizes and eviction cost can be a function of the page size, an underlying assumption remains that the cache space devoted to a page P does not change as the number of requests to P increases. On the contrary, with our problem, n requests to the same page take n units of buffer space. This difference turns out to be fundamental. While we can develop flushing policies analogous to well-studied cache replacement policies, we will see that their performance differs both analytically and experimentally; new policies specialized for flushing are needed.

[5] The buffer in this context should not be confused with the system buffer pool. This buffer batches updates while the buffer pool caches disk pages.

[6] Updates to currently buffered records are simply applied to the buffer, and are not counted as new requests. Therefore, n requests, even if they are for the same page, would take n units of space.

We now present our flushing policies. Here we summarize our theoretical results; see Appendix A for formal statements and proofs. We measure the performance of a flushing policy by its competitive ratio against *OPT*, the optimal offline policy, which knows the entire request sequence in advance. OPT can be implemented by an exponential-time search; the algorithmic details are irrelevant here. (As a side note, the optimal offline cache replacement policy, *furthest-in-future* [4], is not optimal for flushing; see Remark B.6.)

We show that any policy is $O(K)$-competitive (Lemma 2). (Had we been dealing with caching instead, this competitive ratio would have been the best that any deterministic policy can offer.) The most commonly used flushing policy actually does better:

- **Flush-All (ALL).** This policy simply flushes the entire buffer whenever the buffer is full. We show that ALL is $\Omega(\sqrt{K})$- and $O(\sqrt{K}\log K)$-competitive (Theorems 4 and 6).

We can generalize the lower bound above to what we call *c-recent* flushing policies (Definition 3 in appendix), which do not buffer a request for a page if there has been no request for that page during the past cK requests. Clearly, ALL is 1-recent. We show that any c-recent policy is $\Omega(\sqrt{K}/c)$-competitive (Theorem 5).

The next few flushing policies have analogies in caching:

- **Least-Recently-Used (LRU).** This policy always flushes the page whose most recent request is the oldest (among all pages' most recent requests). It is analogous to the classic cache replacement policy of the same name. We show that LRU is $\Omega(\sqrt{K})$-competitive (Corollary 1) by noting that LRU is 1-recent. (Note that for caching, LRU is optimally competitive, with a competitive ratio of K.)
- **Smallest-Page (SP).** This policy always flushes the "smallest" page, i.e., one with the smallest number of currently buffered requests. It is analogous to the LFU (least-frequently-used) cache replacement policy. While LFU is widely used for caching, SP does not make much sense for flushing. Intuitively, SP flushes small pages, but flushing larger ones is more profitable as more requests can be processed with one page write. While SP attempts to preserve large pages, pages have little chance to grow large because they may get flushed when still small.
  We show that SP is $\Theta(K)$-competitive (Lemma 2 and Theorem 8). The example constructed in the proof of Theorem 8 makes the above intuition concrete. (Note that for caching, LFU's competitive ratio is unbounded.)
- **Largest-Page (LP).** This policy always flushes the "largest" page, i.e., one with the largest number of currently buffered requests. It is analogous to the MFU (most-frequently-used) cache replacement policy. LP avoids SP's problem of flushing small pages. On the other hand, LP may flush a page prematurely just because it is currently the largest; however, that page may grow even larger if it not immediately flushed.
  We show that, just like SP, LP is $\Theta(K)$-competitive (Lemma 2 and Theorem 7). The proof of Theorem 7 gives a concrete example of the premature flushing problem.

Next, we present two new polices: the first is a randomized variant of LP, while the second is a novel policy aimed at achieving a fundamentally better competitive ratio than the policies above.

- **Largest-Page-Probabilistically (LPP).** This policy randomly flushes a page with probability proportional to the number of requests currently buffered for this page. It can be seen as a randomization of LP. Intuitively, LPP is designed to avoid the problems of LP and SP: larger pages have a higher chance of being flushed, but all pages have a chance to survive and grow larger. Another attractive feature of LPP is its efficiency of implementation, as we shall see in Section 5.2.
- **Largest-Group (LG).** This policy partitions buffered requests into *groups*: Group i, where $0 \leq i \leq \lfloor \log K \rfloor$, contains a page P if the number of buffered requests for P is in the range $[2^i, 2^{i+1})$. We define the *size of a group* to be the total number of buffered requests for its constituent pages. When the buffer is full, LG flushes the group with the largest size.
  LG is a novel policy designed specifically for the update batching problem. Intuitively, LG's practice of flushing a group at a time offers better protection against an adversary than flushing a page at a time. With $\lfloor \log K \rfloor + 1$ groups, the largest group has at least $\frac{K}{\lfloor \log K \rfloor + 1}$ requests, so LG always flushes a sizable number of requests. Even if LG had chosen a wrong subset of requests to flush, this mistake cannot be repeated until the buffer is full again, which only happens after at least $\frac{K}{\lfloor \log K \rfloor + 1}$ more requests. In contrast, an adversary can more easily penalize policies that may flush a few requests.
  We show that LG has a competitive ratio of $O(\log^3 K)$ (Theorem 9), making it the theoretically best among our policies.

5.2 Implementation

Obtaining Page Identities and Ranges All policies above except ALL require obtaining the page identity and key range for a buffered request. Such information is readily available by executing a "partial" lookup for the requested key in the LAB-tree, without visiting the leaf page containing the key. Only one partial lookup is needed for requests to the same page, because once we obtain page P's range, we can check whether a request refers to P by comparing the requested key with P's range. Since only non-leaf levels are visited, a generic system buffer pool (not to be confused with the update buffer) is effective in reducing I/Os.

LP, SP, and LRU At the time of flush, these policies make one pass over the buffered requests in key order. In the process, we find the identity and range of each requested page P, using one partial lookup (as opposed to one per request to P, as explained above). Remaining details are policy-specific and are given in Remark B.7.

To further reduce page identification cost, we maintain a cache that remembers the identity and range for up to a configurable number of pages. At the next flush, we avoid the cost of identifying such pages. Of course, this page information cache consumes space that could otherwise be devoted to buffering requests, which we account for in our empirical evaluation in Section 5.3.

LPP At the first glance, LPP seems to require knowing the counts of buffered requests for all pages. A far more efficient implementation is possible, however. We simply need to pick one buffered request uniformly at random, find the identity and range of its page, and flush that page (i.e., all buffered requests within its range). Clearly, this implementation picks a page with probability proportional to the number of buffered requests to this page. Remark B.8 gives additional implementation details that enable efficient random sampling and space management.

LG At the time of flush, LG makes one pass over the buffered requests in key order. For each requested page P, we find and record the identity and range of P; using P's range, we count the number of buffered requests to P; using this count, we determine and record the group number of P; finally, we add the count to a running sum that maintains the size of P's group. After this process, we make a second pass to flush the group with the largest size; requests in this group are those with keys that fall within the ranges of its constituent pages. Like LP, SP, and LRU, LG also maintains a page information cache across flushes to reduce the cost of page identification. We populate this cache in the second pass with a subset of pages that are not flushed.

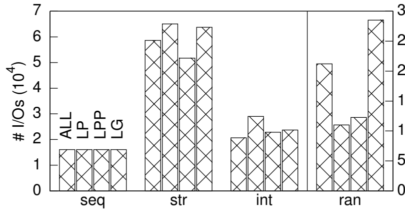


Figure 5: I/O counts of flushing policies.

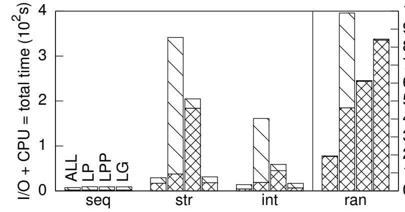


Figure 6: Flushing policies on local drive.

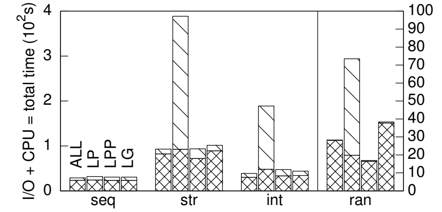


Figure 7: Flushing policies on NFS.

5.3 Experimental Evaluation

We evaluate the flushing policies using *seq*, *str*, *str*, and *ran*, the four insertion patterns from Section 4.3. Here we discuss results for a matrix of size 4000×4000 and a 32MB buffer pool. Updates are buffered in a separate 3MB memory buffer, which holds about 200,000 requests for ALL but fewer for others because of their extra space overhead. LRU, LP, SP, and LG maintain a cache that remembers information about 8000 pages; this space is charged against the update buffer. The scale of these experiments is smaller than those in Section 4.3, but allows us to obtain a complete set of results including those for the most demanding *ran* workloads. Additional results for larger scales are in [18], and they agree with the conclusions below. Because of space constraints, we also omit LRU and SP; they incur unacceptably high CPU cost like LP, which is explained later when we discuss Figure 6.

Figure 5 shows the total number of actual I/Os incurred by each policy (which excludes those serviced by the buffer pool without hitting storage). This metric is unaffected by the characteristics of the underlying storage substrate. LPP, the randomized version of LP, turns out a winner: across all patterns, LPP is either the best or comes close to the best. ALL is noticeably worse than LPP for *str* (13% more I/Os), and much worse for *ran* (73% more I/Os). LG, despite its attractive worst-case theoretical guarantee, fails to distinguish itself for these common insertion patterns. LP has reasonable I/O counts, but we will soon see its crippling disadvantage.

As storage substrates grow more diverse and sophisticated, the relationship between I/O count and running time has become increasingly dependent on the system specifics. Therefore, we compare running time for two different storage substrates: `ext2` on a local hard drive, and NFS over network-attached storage (NAS).

Figure 6 summarizes the results for the local hard drive. Here ALL really shines. A closer inspection reveals that LPP suffers from random I/Os because of its inherent randomness; its I/O times are higher than ALL even when its I/O counts are much lower. The high CPU overhead destroys LP, because at the time of each flush, it needs to scan the entire buffer and identify all pages requested. Although the buffer pool is efficient in reducing I/O needed for page identification (as evidenced in Figure 5), the CPU overhead remains. Another interesting observation is the stark contrast between LP's high CPU overhead and LG's low CPU overhead, since LG's flushing procedure seems more costly than LP's. However, by flushing only large groups, LG flushes much less frequently than LP, so the amortized cost of its procedure becomes much lower.

On the other hand, results for NAS through NFS (Figure 7) tell a very different story. Here, the I/O component of the total time is more consistent with the I/O count in Figure 5. For this reason, and because of LPP's low CPU overhead, LPP's I/O count advantage over ALL carries over, and LPP becomes the overall winner.

6 Conclusion

We have presented LAB-tree as a solution for storing arrays on disk to support scalable analysis. It uses linearization to provide flexible array layouts, and has a dynamic leaf format that adapts to varying sparsity across space and time. Experiments on common workloads and real data confirm its advantage over B-tree and directly addressable files. We have also called into question the standard B-tree strategy for splitting overflowing leaves and the common flush-all policy for update batching. Based on our theoretical analysis and empirical evaluation, we conclude with some recommendations. 1) We believe split-aligned should replace split-in-middle as the choice of splitting strategy for array data, because of its good theoretical properties and practical performance. 2) For update batching, when the difference between random and sequential writes is obscured (e.g., log-structured file system) or nonexistent (e.g., phase-change memory), we recommend flush-largest-page-probabilistically, with fewer I/Os and low CPU overhead. On conventional hard drives, however, the best bet remains flush-all, for which we also prove a reasonable competitive ratio.

References

[1] Baumann. Management of multidimensional discrete data. *VLDB J.*, 3(4):401–444, 1994.

[2] Bayer. The universal B-tree for multidimensional indexing: General concepts. In *Worldwide computing and its applications*, pages 198–209, 1997.

[3] Bayer and Unterauer. Prefix B-trees. *TODS*, 2(1):11–26, 1977.

[4] Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Sys. J.*, 5(2):78–101, 1966.

[5] Chen et al. Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software*, 35:22:1–22:14, 2008.

[6] Cohen et al. MAD skills: New analysis practices for big data. In *VLDB*, pages 1481–1492, 2009.

[7] Cudre-Mauroux, Wu, and Madden. The case for RodentStore, an adaptive, declarative storage system. In *CIDR*, 2009.

[8] Davis and Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, To appear.

[9] Küspert. Storage utilization in B*-trees with a generalized overflow technique. *Acta Inf.*, 19:35–55, 1983.

[10] Lang, Driscoll, and Jou. Batch insertion for tree structured file organizations - improving differential database reprensentation. *Inf. Syst.*, 11(2):167–175, 1986.

[11] Lohman and Muckstadt. Optimal policy for batch operations: Backup, checkpointing, reorganization, and updating. *TODS*, 2(3):209–222, 1977.

[12] Lomet. The evolution of effective B-tree: Page organization and techniques: A personal account. *SIGMOD Rec.*, 30(3):64–69, 2001.

[13] Marathe and Salem. Query processing techniques for arrays. *VLDB J.*, 11(1):68–91, 2002.

[14] Ordonez and García-García. Vector and matrix operations programmed with UDFs in a relational DBMS. In *CIKM*, pages 503–512, 2006.

[15] SciDB. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.

[16] Sleator and Tarjan. Amortized efficiency of list update and paging rules. *CACM*, 28(2):202–208, 1985.

[17] Zhang, Herodotou, and Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *CIDR*, 2009.

[18] Zhang, Munagala, and Yang. Storing matrices on disk: Theory and practice revisited. Technical report, Duke University, 2011. `http://www.cs.duke.edu/dbgroup/papers/2011-ZhangMunagalaYang-matrix_store.pdf`.

APPENDIX

A Theorems and Proofs

Theorem 1. *No deterministic online local splitting strategy has a competitive ratio less than* 2.

Proof. Given any splitting strategy Σ, we construct an insertion sequence that results in a tree with overall density of at most $1/2$, i.e., with at least twice the number of leaves produced by the optimal offline algorithm. We start by inserting any $\kappa + 1$ records, causing the first split. There are two cases. 1) If any of the two result leaves has a range with length no greater than κ, we mark both nodes *inactive*; 2) otherwise, we mark the result leaf with fewer records *inactive* and the other one *active*. After the split, we pick any active leaf and keep inserting records into it until the next split. The process is repeated until there is no active leaf left.

In the end, all leaves are inactive. Those generated by Case 2 all have density no greater than $1/2$. Each inactive leaf generated by Case 1 is paired with exactly one other inactive leaf. These two leaves are resulted from splitting a leaf and have not be inserted into since. Thus, their combined density is (arbitrarily close to) $1/2$. Therefore, the overall density of the tree is at most $1/2$. □

Lemma 1. *Every leaf produced by a no-dead-space splitting strategy must have a range whose length is divisible by κ; i.e., all splitting points picked are multiples of κ.*

Proof. For brevity, if a leaf has a range length not divisible by κ, we call the leaf *misaligned*. No matter how we choose the splitting point, if the original node is misaligned, at least one of the resulting leaves after the split is misaligned. If a splitting strategy ever produces any misaligned leaf that is not full, we keep inserting into it, and continue inserting into any misaligned leaf subsequently generated, until a misaligned leaf ℓ with range length less than κ is produced. At this point, no future insertion sequence can eliminate the dead space in ℓ. Therefore, the splitting strategy does not have the no-dead-space property.

For an array with range $[0, m\kappa)$ for some integer m, all splitting points picked by a no-dead-space splitting strategy must be multiples of κ because the range of the first leaf starts with 0. □

Theorem 2. *Any no-dead-space splitting strategy has a competitive ratio of at least* 3.

Proof. Given any no-dead-space splitting strategy, we construct an insertion sequence that results in at least three times the number of leaves produced by the optimal offline algorithm. Suppose the tree's key domain spans range $[0, m\kappa)$. We call each interval $[i\kappa, (i+1)\kappa)$ a *unit interval*; there are m unit intervals. Without loss of generality, assume $\kappa + 1 = 3k$ where $k \in \mathbb{N}$.

Starting with an empty tree, insert k records each into the first (0-th), last ($(m-1)$-th), and middle ($(\frac{m-1}{2})$-th) unit intervals, causing the first split. By Lemma 1, records in the same unit interval will never be separated. It is thus clear that after the first split, one leaf contains k records and the other contains $2k$. We leave the smaller leaf intact. We call the larger leaf ℓ. The range of ℓ covers $\frac{m+1}{2}$ unit intervals, the first and last of which contain k records each. We then insert k records into the unit interval right in the middle of ℓ, resulting in the second split with the configuration as the first, except that total range is halved. This process can be repeated recursively until the larger leaf contains only 2 unit intervals. In the end, all except one leaf contain $k = \frac{\kappa+1}{3}$ records each. Therefore, the number of leaves is (arbitrarily close to) three times that of the optimal offline algorithm. □

Theorem 3. *Split-aligned has a competitive ratio of* 3.

Proof. Given any insertion sequence on an initially empty tree T, we construct a *split tree* S, which captures the history of node splits. We maintain a bijection f between T's leaves (including those that *were* leaves at one point but were later split and thus do not exist in T any more) and S's nodes by the following procedure: Initially T contains only an empty leaf ℓ and S contains a single node, $f(\ell)$; whenever a leaf ℓ of T splits into ℓ_1 and ℓ_2, we create two new nodes $f(\ell_1)$ and $f(\ell_2)$ in S and add them as $f(\ell)$'s left and right children, respectively. By construction, for any node x in T, x is a leaf in T iff $f(x)$ is a leaf in S. To simplify notation, we will use x to mean $f(x)$ when there is no confusion. For any node x in S, we denote its parent in S by $p(x)$, and its sibling in S by $s(x)$. In both T and S, if x is a leaf, we denote the leaf to its left by $\alpha(x)$, and the leaf to its right by $\beta(x)$.

In the following, we show that all leaves of T (or S) can be put into groups so that each group has density at least $\frac{1}{3}$. Therefore, for any insertion sequence, the number of leaves generated by the split-aligned is at most three times of that generated by the optimal offline strategy, establishing a competitive ratio of 3. The tightness is given by Theorem 2.

We group leaves of S as follows:

- **Case A.** For any leaf x in S whose sibling is also a leaf, we put x and its sibling into one group. Such a group has at least $\kappa + 1$ records, so the density is at least $\frac{1}{2}$.
- **Case B.** For any leaf x in S whose sibling is not a leaf, if $\rho(x) \geq \frac{1}{3}$, we put x in one group by itself.
- **Case C.** For any leaf x in S whose sibling is not a leaf, if $\rho(x) < \frac{1}{3}$, we put x to an existing group. Specifically, if x is the left (or right) child of its parent in S, we add it to a group to its right (or left, respectively).

It remains to be shown that in Case C above, adding x to an existing group never decrease the group's density to below $\frac{1}{3}$.

Without loss of generality, assume x is a left child. Consider the point when $p(x)$ was split into x and $s(x)$. Suppose $p(x)$ has range $[h\kappa, j\kappa)$ and the splitting point was $i\kappa$ ($h < i < j$). Let ϱ be the density of x right after this split; i.e., inside $p(x)$ before the split, interval $[h\kappa, i\kappa)$ contained $\varrho\kappa$ records. Clearly, $\varrho \leq \rho(x) < \frac{1}{3}$. We define the *companion interval* of x, denoted $C(x)$, to be the unit interval $[i\kappa, (i+1)\kappa)$, i.e., one that is adjacent to x's range and anchored to the splitting point. Note that $C(x)$ must contain the $\varrho\kappa$-th to the $((1-\varrho)\kappa - 1)$-th records in $p(x)$ before the split. Had there existed a possible splitting point $i'\kappa > i\kappa$ between the $\varrho\kappa$-th and the $((1-\varrho)\kappa - 1)$-th records of $p(x)$, $i'\kappa$ would have been a better splitting point than $i\kappa$ according to split-aligned, as two resulting leaves would have been more balanced in their numbers of records. As a unit interval, $C(x)$ will never be split, so $C(x)$ always contains at least $(1 - 2\varrho)\kappa > \frac{1}{3}\kappa$ records. Figure 8 illustrates the definition of companion interval.

After the split, $s(x)$ contained $C(x)$. In S, $\beta(x)$ is $s(x)$'s left-most descendant, which must contain $C(x)$ as its left-most unit interval; thus, $\beta(x)$ has more than $\frac{1}{3}\kappa$ records. We have two cases.

Case 1. If $s(\beta(x))$ is not a leaf, then $\beta(x)$ is in a group G just by itself (Case B); we simply add x to G. The other nodes that can possibly join G are $\beta(\beta(x))$ and $\alpha(x)$, but we argue that they will not. As a left child, $\beta(\beta(x))$ can only join the group to its right. If $\alpha(x)$ is a right child, it can only join the group to its left. If $\alpha(x)$ is a left child, then x must be $s(\alpha(x))$'s leftmost descendant and thus contain $C(\alpha(x))$. However, for $\alpha(x)$ to be added to an existing group, we must have $\rho(\alpha(x)) < \frac{1}{3}$ and $C(\alpha(x))$ must contain more than $\frac{1}{3}\kappa$ records, contradicting the fact that $\rho(x) < \frac{1}{3}$.

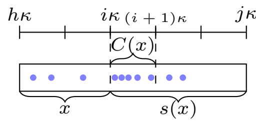


Figure 8: Companion interval.

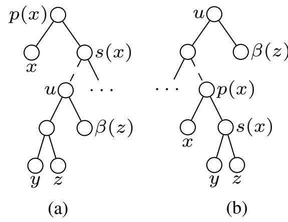


Figure 9: Cases 2a and 2b in the proof of Theorem 3.

Therefore, the density of G, which contains only x and $\beta(x)$, is at least $\frac{(\varrho+1-2\varrho)\kappa}{2\kappa} > \frac{1}{3}$.

Case 2. If $s(\beta(x))$ is a leaf, then $\beta(x)$ and $s(\beta(x))$ are already in a group G (Case A). We add x to G and claim that no other node can join G. Let $y = \beta(x)$ and $z = s(y) = \beta(y)$; i.e., G contains x, y, and z. The other nodes that can possibly join G are $\alpha(x)$ and $\beta(z)$. As a right child, $\alpha(x)$ cannot join a group to its right. Now consider $\beta(z)$. If $\beta(z)$ is a left child, it will never be added to a group to its left. If $\beta(z)$ is a right child, there are two cases based on how long the path from $s(x)$ to y is. Note that this path consists of left branches only.

- **Case 2a.** If the path length is at least 2, then $\beta(z)$ is the right child of $u = p(p(z))$ (Figure 9(a)). Consider the time when u split. Obviously x was created before u, so u contained $C(x)$. For $\beta(z)$ to be split off, $\beta(z)$ had to contain no fewer records than $C(x)$; otherwise the split would not be the most balanced one. Therefore, $\rho(\beta(z)) > \frac{1}{3}$, and hence $\beta(z)$ will not be added to any existing group.
- **Case 2b.** If the path length from $s(x)$ to y is 1, then $p(y) = p(z) = s(x)$. In this case, $\beta(z)$ is found by traversing up the tree from z until the first left branch is taken, to $u = p(\beta(z))$, and then taking the sibling right branch down (Figure 9(b)). It is obvious that $\beta(z)$ was created before x. Note that all nodes along the path from $s(\beta(z))$ to z contain $C(\beta(z))$ as their rightmost unit interval. Consider $p(x)$ on this path, which contained at least three unit intervals (because of its three descendants). In order for $\beta(z)$ to be added to G, we must have $\rho(\beta(z)) < \frac{1}{3}$. In that case, however, $C(\beta(z))$, or $p(x)$'s rightmost unit interval, would contain more than $\frac{1}{3}\kappa$ records, contradicting the fact that x, with fewer than $\frac{1}{3}\kappa$ records, was split off $p(x)$. Therefore, $\beta(z)$ will not be added to any existing group.

To conclude Case 2, G contains x, y, and z, and has density at least $\frac{\varrho\kappa+\kappa}{3\kappa} > \frac{1}{3}$. □

Lemma 2. *Any flushing policy is $O(K)$-competitive.*

Proof. For any page P, OPT flushes P at least once per K requests for P. Any policy flushes P at most once per request for P. □

Theorem 4. *ALL is $\Omega(\sqrt{K})$-competitive.*

Proof. Let $R = \lfloor\sqrt{K}\rfloor$. We construct a request sequence consisting of $\frac{R}{2}$ phases, each with K requests. Each such phase has one request for each of pages $P_1, P_2, \ldots, P_R$, plus $K - R$ requests for page P_0. ALL incurs a cost of $R+1$ per phase.

On the other hand, a better policy Π would keep all requests for $P_1, \ldots, P_R$ until the end of the last phase, and in the meantime, flush P_0 as needed. The number of requests for $P_1, \ldots, P_R$ in the buffer increases up to $\frac{R}{2}R \le \frac{K}{2}$, so at least $\frac{K}{2}$ space is always available for buffering P_0 requests. Therefore, Π needs to flush P_0 at most twice per phase, and flush $P_1, \ldots, P_R$ once at the end of the $R/2$ phases. Π's amortized cost per phase is $\le 2 + \frac{R}{R/2} = 4$. □

Definition 3 (c-recency)**.** *A flushing policy is* c-recent *if it has the following property: If there has been no request for page P among the past cK requests, then no request for P is currently buffered.*

Theorem 5. *Any c-recent flushing policy is $\Omega(\sqrt{K}/c)$-competitive.*

Proof. Consider the request sequence from the proof of Theorem 4. We modify it as follows. For each phase, we add cK requests for page P_0 at the end of the phase. These new requests would force any c-recent algorithm to flush $P_1, P_2, \ldots, P_R$, incurring a cost of R per phase for these pages.

On the other hand, a better policy Π would keep flushing P_0 as needed, incurring at most $2c$ flushes of P_0 per phase since at least $K/2$ space is available for buffering P_0 requests. □

Corollary 1. *LRU is $\Omega(\sqrt{K})$-competitive.*

Proof. By Theorem 5, it suffices to show that LRU is 1-recent. For any page P currently buffered, consider the K requests immediately following the most recent request for P. If none of these K requests are for P, LRU must flush them later than P. P cannot be buffered after these K requests because that would require the buffer to hold P plus the K requests, exceeding its capacity. □

Theorem 6. *ALL is $O(\sqrt{K}\log K)$-competitive.*

Proof. Divide the request sequence into phases of length K. Without loss of generality, assume that the total number of requests is a multiple of K. Suppose there are m pages. Let m-dimensional vector $\mathbf{r}^{(t)}$ denote the collection of requests in Phase t, where the i-th component of the vector, denoted $r_i^{(t)}$, specifies the number of requests for P_i.

We consider the behavior of a policy Π. Π has a buffer of size $2K$ and mimics OPT as follows. Let vector $\mathbf{s}^{(t)}$ denote the state of OPT's buffer at the beginning of Phase t, where $s_i^{(t)}$ specifies the number of requests for P_i buffered by OPT at that time. Π buffers all these requests (using at most K space) throughout Phase t, together with all requests (using K space) received during Phase t. At the end of Phase t, Π flushes whatever requests that OPT has flushed during Phase t. Clearly, Π and OPT incur the same cost over the entire request sequence.

Define a potential function over the current buffer state $\mathbf{s}$ as: $\Phi(\mathbf{s}) = \sum_{i=1}^{m} \ln(1+s_i)$. Let Δ be the total potential increase due to incoming requests over the course of executing Π on the entire input sequence. Each flush of Π involves at most $2K$ requests and lowers the potential by at most $\ln(2K+1)$. The potential is 0 at the beginning and the end of the entire request sequence. Therefore, $\Delta/C^{\text{OPT}} = O(\log K)$, where C^{OPT} is the same as the total number of flushes by Π.

Consider the change in potential in Phase t. We divide the requests in this phase into two groups:

- Requests for *cold pages*, where a page P_i is *cold* in Phase t if $r_i^{(t)} \ge 1$ and $\ln\left(\frac{1+s_i^{(t)}+r_i^{(t)}}{1+s_i^{(t)}}\right) < \frac{1}{\sqrt{K}}$; i.e., the total potential increase in Phase t due to P_i requests is less than $1/\sqrt{K}$.
- Requests for *hot pages*, where the total potential increase in Phase t due to requests to each hot page is at least $1/\sqrt{K}$.

Let C_t^{ALL} denote the cost of ALL incurred in Phase t. Clearly, $C_t^{\text{ALL}} = q_c + q_h$, where q_c is the number of cold pages and q_h is

the number of hot pages in Phase t. Note that for a cold page P_i,

$$\frac{1}{\sqrt{K}} > \ln\left(\frac{1+s_i^{(t)}+r_i^{(t)}}{1+s_i^{(t)}}\right) = \ln\left(1+\frac{r_i^{(t)}}{1+s_i^{(t)}}\right)$$
$$> \frac{r_i^{(t)}}{1+s_i^{(t)}} \Big/ \left(1+\frac{r_i^{(t)}}{1+s_i^{(t)}}\right) = \frac{r_i^{(t)}}{1+s_i^{(t)}+r_i^{(t)}},$$

which implies $s_i^{(t)} > (\sqrt{K}-1)r_i^{(t)} - 1$. It follows that $q_c < \frac{K}{\sqrt{K}-2}$, because $K \geq \sum_{P_i \text{ is cold}} s_i^{(t)} > (\sqrt{K}-1)\sum_{P_i \text{ is cold}} r_i^{(t)} - q_c \geq (\sqrt{K}-2)q_c$. Let Δ_t denote the total potential increase due to incoming requests in Phase t. We have $\Delta_t \geq q_h/\sqrt{K}$. At the same time, note that there exists at least one page P_j with $r_j^{(t)} \geq \max(1, s_j^{(t)})$ (otherwise, $\sum_{i=1}^m r_i^{(t)} < \sum_{i=1}^m s_i^{(t)} = K$, a contradiction), so

$$\Delta_t \geq \ln\left(\frac{1+s_j^{(t)}+r_j^{(t)}}{1+s_j^{(t)}}\right) \geq \ln\left(\frac{1+s_j^{(t)}+\left(1+s_j^{(t)}\right)/2}{1+s_j^{(t)}}\right)$$
$$\geq \ln 1.5 > 0.4.$$

Therefore,

$$\frac{C_t^{\text{ALL}}}{\Delta_t} \leq \frac{q_c+q_h}{(0.4+q_h/\sqrt{K})/2} = 2\sqrt{K}\cdot\frac{q_c+q_h}{0.4\sqrt{K}+q_h}$$
$$< 2\sqrt{K}\cdot\frac{\frac{K}{\sqrt{K}-2}+q_h}{0.4\sqrt{K}+q_h} = O(\sqrt{K}).$$

Finally, let C^{ALL} denote the cost of ALL over the entire request sequence. $C^{\text{ALL}}/\Delta = (\sum_t C_t^{\text{ALL}})/(\sum_t \Delta_t) = O(\sqrt{K})$. We have already shown $\Delta/C^{\text{OPT}} = O(\log K)$, so $C^{\text{ALL}}/C^{\text{OPT}} = O(\sqrt{K}\log K)$. □

Theorem 7. *LP is $\Omega(K)$-competitive.*

Proof. Consider the request sequence

$$P_1, P_2, P_3, \ldots, P_{K-2}, P_0, P_0, P_0^*,$$

where P_0^* denotes repeating P_0 requests. After the K-th request (the second P_0), the buffer contains two P_0 requests and one request for every other page, so LP flushes P_0. Subsequently, LP incurs one unit of cost every two new P_0 requests.

A better policy Π would be to first flush $P_1, P_2, \ldots, P_{K-2}$ after the K-th request. Subsequently, Π would buffer P_0 and flush when needed, incurring one unit of cost every K new P_0 requests. □

Theorem 8. *SP is $\Omega(K)$-competitive.*

Proof. Let $K = 3k+1$. Consider the request sequence

$$P_1, P_1, P_1, P_2, P_2, P_2, \ldots, P_k, P_k, P_k, P_0, P_0^*.$$

After the K-th request (the first P_0), the buffer contains one P_0 request and three requests for every other page, so SP flushes P_0. Subsequently, SP incurs one unit of cost for each new P_0 request.

A better policy Π would be to first flush $P_1, P_2, \ldots, P_k$ after the K-th request. Subsequently, Π would buffer P_0 and flush when needed, incurring one unit of cost every K new P_0 requests. □

Lemma 3. *If OPT is given a buffer of size K/c (where $c > 1$) instead of K, its cost increases by at most a factor of $2c\lceil\log K\rceil$.*

Proof. Given the behavior of OPT using a buffer of size K, we design a policy Π using a buffer of size K/c as follows. Suppose OPT flushes x requests for a page P. We divide the period between this flush and the previous flush of P into phases, according to how many P requests have been buffered by OPT:

- The first phase is when this number is within $[1, 2^\sigma]$, where $\sigma = \lfloor\log c\rfloor + 1$;
- In each subsequent phase this number is within $[2^i+1, 2^{i+1}]$, for $\sigma \leq i \leq \lceil\log x\rceil - 1$.

For the first phase, Π would reserve no buffer space for P, and simply flush every request for P immediately; there are at most $2^\sigma \leq 2c$ flushes. For the phase corresponding to $[2^i+1, 2^{i+1}]$, Π would reserve $\lfloor(2^i+1)/c\rfloor$ buffer space for P, and flush P whenever the reserved capacity is reached or at the end of the phase; there are at most $\lceil\frac{2^{i+1}-2^i}{\lfloor(2^i+1)/c\rfloor}\rceil < 2c$ flushes because $i \geq \sigma$.[7] Since the number of phases is at most $\lceil\log x\rceil \leq \lceil\log K\rceil$, Π does at most $2c\lceil\log K\rceil$ flushes for each flush of P by OPT.

Finally, it is easy to see that Π never uses more than K/c space. During each phase, OPT spends more space on P than it does at the beginning of the phase, while Π uses no more than $1/c$ of that amount. Therefore, Π uses no more than $1/c$ of the space used by OPT at any time. □

Theorem 9. *LG is $O(\log^3 K)$-competitive.*

Proof. Divide the request sequence into $R = \lfloor\log K\rfloor + 1$ subsequences, one for each group in LG. Subsequence $\mathcal{S}_i$ contains all requests that are flushed by LG as part of Group i.

For each $i \in [0, R)$, our first step is to compare C_i^{LG}, the number of flushes of Group i by LG, against C_i^{OPT}, the number of flushes incurred by running OPT on $\mathcal{S}_i$ with a buffer of size $\lfloor\frac{K}{3R}\rfloor$. Divide $\mathcal{S}_i$ into phases separated by flushes of Group i by LG. Consider any such phase. Let r denote the number of requests in this phase. We have $r \geq \frac{K}{R}$ because there are R groups and LG always flushes the largest group. Let q denote the number of distinct pages requested in this phase. $C_i^{\text{LG}} = q$. OPT has to flush a page P at least once in this phase if OPT is unable to buffer all requests to P in this phase. Recall from the definition of Group i that the number of requests per page is in the range $[2^i, 2^{i+1})$. With $\lfloor\frac{K}{3R}\rfloor$ space, OPT can buffer all requests for no more than $\lfloor\frac{K}{3R}\rfloor/2^i$ pages. Therefore,

$$\frac{C_i^{\text{OPT}}}{C_i^{\text{LG}}} \geq \frac{q-\lfloor\frac{K}{3R}\rfloor/2^i}{q} \geq 1-\frac{(\frac{r}{3})/2^i}{q} \quad \text{by } r \geq \frac{K}{R}$$
$$\geq 1-\frac{(\frac{r}{3})/2^i}{\frac{r}{2^{i+1}-1}} \quad \text{by } r \leq (2^{i+1}-1)q$$
$$> 1/3.$$

Let C^{LG} denote the total cost of LG, and let C^{OPT} denote the cost of running OPT with a buffer of size K over the entire request sequence. Next, we will show that $C_i^{\text{OPT}}/C^{\text{OPT}} = O(R^2)$, so

$$\frac{C^{\text{LG}}}{C^{\text{OPT}}} = \frac{\sum_{i=0}^{R-1} C_i^{\text{LG}}}{C^{\text{OPT}}} < 3\sum_{i=0}^{R-1}\frac{C_i^{\text{OPT}}}{C^{\text{OPT}}} = \sum_{i=0}^{R-1} O(R^2) = O(R^3),$$

completing the proof. To this end, note that $C_i^{\text{OPT}_K}$, the cost of OPT on $\mathcal{S}_i$ with K space, must be no more than C^{OPT}, where OPT runs on a strictly bigger sequence. Therefore, by Lemma 3,

$$\frac{C_i^{\text{OPT}}}{C^{\text{OPT}}} < \frac{C_i^{\text{OPT}}}{C_i^{\text{OPT}_K}} = O(R\lceil\log K\rceil) = O(R^2).$$

□

[7] If $i \geq \log c + 1$, then $2^i > 2c - 1$, and we can show $\lceil\frac{2^{i+1}-2^i}{\lfloor(2^i+1)/c\rfloor}\rceil < \frac{2^{i+1}-2^i}{(2^i+1)/c-1} + 1 < 2c$ (details omitted). If $\lfloor\log c\rfloor + 1 = \sigma \leq i < \log c + 1$, we have $2^i \in (c, 2c)$, so $\lfloor(2^i+1)/c\rfloor \geq 1$ and the inequality follows.

B Additional Remarks

Remark B.1 (Deriving $\tau(x)$ for Split-Defer-Next; Section 4.1) Let $k_1 = 1 + \arg\max_j i_j < x$ and $k_2 = \kappa + 1 - k_1$ be the numbers of records initially in ℓ_1 and ℓ_2, respectively. Let $b_1 = \kappa - k_1$ and $b_2 = \kappa - k_2$ be the numbers of free storage slots initially in ℓ_1 and ℓ_2, respectively. Let $d_1 = x - l - k_1$ and $d_2 = u - x - k_2$ be the numbers of keys initially missing from the ranges of ℓ_1 and ℓ_2, respectively. If $d_1 \leq b_1$ and $d_2 \leq b_2$, $\tau(x) = \infty$; otherwise,

$$\tau(x) = \sum_{k \in [b_1, b_1+b_2]} (k+1) \cdot \frac{\binom{d_1}{b_1}\binom{d_2}{k-b_1}}{\binom{d_1+d_2}{k}} \cdot \frac{d_1 - b_1}{d_1 + d_2 - k} + \sum_{k \in [b_2, b_1+b_2]} (k+1) \cdot \frac{\binom{d_2}{b_2}\binom{d_1}{k-b_2}}{\binom{d_1+d_2}{k}} \cdot \frac{d_2 - b_2}{d_1 + d_2 - k}.$$

This expectation is calculated over all possible sequences of $d_1 + d_2$ insertions into $[l, u)$. We give the intuition behind the first summation; the second summation is analogous. Each summand in the first summation corresponds to the case that ℓ_1 splits after $k+1$ insertions into $[l, u)$. For this case to happen, the first k insertions must have completely filled up ℓ_1's space, and the last insertion still goes into ℓ_1's range. The second term in the summand, $\binom{d_1}{b_1}\binom{d_2}{k-b_1}/\binom{d_1+d_2}{k}$, calculates the fraction of all insertion sequences whose first k insertions fill up ℓ_1 completely. The third term, $\frac{d_1-b_1}{d_1+d_2-k}$, further calculates the fraction of these insertion sequences whose $(k+1)$-th insertion goes into ℓ_1's range.

The above formula can be further simplified as

$$\tau(x) = \frac{\sum_{k \in [0, b_1+b_2]} (k+1) \left(\binom{k}{b_1}\binom{d_1+d_2-k-1}{d_1-b_1-1} + \binom{k}{b_2}\binom{d_1+d_2-k-1}{d_2-b_2-1} \right)}{\binom{d_1+d_2}{d_1}}.$$

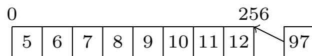
Remark B.2 (Formula for Setting the Splitting Point for Split-Balanced-Ratio; Section 4.1) Given an overflowing leaf with range $[l, u)$ and keys $i_0, i_1, \ldots, i_\kappa$, this strategy sets the splitting point $x = i_k$, where

$$k = \arg\max_j \left(\min \left(\frac{\kappa - j}{(i_j - l) - j}, \frac{\kappa - (\kappa + 1 - j)}{(u - i_j) - (\kappa + 1 - j)} \right) \right).$$

Remark B.3 (Handling Deletions; Section 4.1) For all splitting strategies discussed in Section 4.1, it is possible to devise a strategy for merging LAB-tree leaves analogous to that for B-tree. On the other hand, except for split-in-middle, there is no analogy of stealing from an adjacent leaf, because it might undo the careful choice of leaf range endpoints.

For our LAB-tree implementation, we in fact adopt a simpler approach taken by many practical B-tree implementations. Namely, we do not merge index nodes when underflow occurs; a node is deleted only when it is completely empty. The competitive ratio for space consumption will be broken for workloads involving deletions, but it is acceptable because deletions are rare in practice.

Remark B.4 (Modifying Splitting Strategies for Dynamic Leaf Format; Section 4.2) For split-aligned, we require the splitting point to be a multiple of κ_d. Other necessary modifications are not difficult to devise, but care is needed to cover all possible cases. Because of limited space, we will illustrate just one intricacy with an example. With $\kappa_s = 4$ and $\kappa_d = 8$, consider the following overflowing dense leaf upon the insertion of key 97:



Without modification, split-aligned would choose 8 (a multiple of κ_d) as the splitting point. However, the result right leaf cannot store all of 8, 9, 10, 11, 12, and 97, with either dense or sparse format. Hence, it is necessary to further modify split-aligned to rule out infeasible splitting points. In this case, 96 will be chosen instead.

Remark B.5 (Experimental Setup; Sections 4.3 and 5.3) We ran all our experiments on a Dell Optiplex 960 running Fedora 14 (kernel version 2.6.35.11), with Intel Core 2 Duo E8500 3.16GHz CPU, 8GB of memory, and a 160GB SATA hard drive. We used the `systemtap` tool to measure I/O and time costs, with `systemtap`'s built-in system call probes as well as some user-space probes in our code. We verified that instrumentation overhead was negligible. To make it easier to understand results, we used the `ext2` file system (unless otherwise noted), as it does not have journaling that would unnecessarily complicate result interpretation; we also implemented our own buffer pool manager with LRU and 8KB-sized pages, and turned off file system caching using the `O_DIRECT` flag.

Remark B.6 (Furthest-in-Future is not OPT; Section 5.1) Consider the following request sequence (both subscripts and superscripts are used to differentiate pages): $K-1$ requests to P_0, then $P_1^1, P_2^1, \ldots, P_{K-1}^1, P_0, P_{K-1}^1, P_{K-1}^1, \ldots, P_1^1$, another $K-1$ requests to P_0, then $P_1^2, P_2^2, \ldots, P_{K-1}^2, P_0, P_{K-1}^2, P_{K-2}^2, \ldots, P_1^2$, etc. Because of space constraints, we will only sketch out the initial and critical steps of furthest-in-future and a better policy; see [18] for details. In its initial steps, furthest-in-future will try to keep all the P_0's, forcing the ensuing P_j^1's out one after another and causing them to miss the opportunity to be batched with their second requests. A better policy would flush the P_0's upfront to make space for P_j^1 pairs, more profitable to flush than individual P_j^1's.

Remark B.7 (Additional Implementation Details for LP, SP, and LRU; Section 5.2) For LP and SP, for each requested page P we identify, we count the number of buffered requests to P using P's range, and move to the first request to a different page. We remember the page with the largest or smallest (for LP or SP, respectively) number of buffered requests encountered so far. LP can terminate the process early once we know that the largest page has been found, e.g., when its number of requests is no less than the number of remaining requests to be examined. SP can terminate early as soon as it finds a page with a single request.

For LRU, we record the time when each buffered request entered the buffer. This information requires additional space and therefore reduces the number of requests that can be buffered. However, this information can be compressed at the expense of accuracy. During the pass over the buffered requests, for each requested page P we identify, we scan the buffered requests in P's range and determine the last time when P is requested. We remember the page with the earliest such time among the pages we have encountered.

Remark B.8 (Additional Implementation Details for LPP; Section 5.2) We store all buffered requests in an array $\mathcal{A}$ in memory in no particular order. We maintain an ordered search tree $\mathcal{T}$ on top of $\mathcal{A}$, which allows us to find the locations of requests in $\mathcal{A}$ given a request key range. To pick a page to flush, we simply pick a random location in $\mathcal{A}$ (which is full at the time of the flush). With the key of the buffered request at this location, we obtain the key range for the page to be flushed, using a partial LAB-tree lookup. We then use this key range to search $\mathcal{T}$ for all buffered requests that we need to flush. As we flush them, we chain the reclaimed locations in $\mathcal{A}$ into a linked list, whose links can be stored in $\mathcal{A}$ itself. This linked list is then used for adding new requests to $\mathcal{A}$ efficiently.

Acknowledgment

Y. Zhang and J. Yang are supported by the NSF award IIS-0916027 and an Innovation Research Program Award from HP Labs. K. Munagala is supported by an Alfred P. Sloan Research Fellowship and by NSF via CAREER award CCF-0745761 and grants CCF-1008065 and IIS-0964560.