T. Grust et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2019),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2019 77

Waves of misery after index creation

Nikolaus Glombiewski[1] Bernhard Seeger[2] Goetz Graefe[3]

Abstract: After creation of a new b-tree, the ordinary course of database updates and index maintenance causes waves of node splits. Thus, a new index may at first speed up database query processing but then the first "wave of misery" requires effort for frequent node splits and imposes spikes of buffer pool contention and of I/O. Waves of misery continue over multiple instances although eventually the waves widen, flatten, and spread further apart. Free space in each node left during index creation fails to prevent the problem; it merely delays the onset of the first wave. We have found a theoretically sound way to avoiding these waves of misery as well as some simple and practical means to reduce their amplitude to negligible levels. Experiments demonstrate that these techniques are also effective. Waves of misery occur in databases and in key-value stores, in primary and in secondary b-tree indexes, after load operations, and after b-tree reorganization or rebuild. The same remedies apply with equal effect.

Keywords: Indexing, Bulk Loading, B-tree

1 Introduction

The purpose of adding an index to a database table is to improve the performance of query processing. Unfortunately, after an initial "honey moon" of using a new index and of enjoying fast queries, the index may grow and require many node splits – thus creating a spike in buffer pool contention and I/O activity. In other words, the index may actually reduce query performance or at least fail to achieve the expected performance. Consider the following example. First, a new secondary b-tree index enables fast look-ups and fast ordered scans. In order to absorb updates without node splits, each node might start with 10% free space – often a parameter of index creation. However, once the table and the secondary index approach 10% growth, many of the b-tree leaf nodes require splitting. Thus, there is a wave of split activity with contention for the data structures for free space management, for the buffer pool, and for the I/O devices. Once most of the original index leaves are split, this activity subsides until the table and index grow above two times the original contents, whereupon another wave of splits happens. During each wave of splits, both update and query performance suffer, as do buffer pool contention and space utilization

[1] Department of Mathematics and Computer Science, University of Marburg, 35032 Marburg, Germany, glombien@mathematik.uni-marburg.de
[2] Department of Mathematics and Computer Science, University of Marburg, 35032 Marburg, Germany, seeger@mathematik.uni-marburg.de
[3] Google Inc., Madison WI, USA, goetzgraefe@gmail.com

doi:10.18420/btw2019-06

in memory and on storage. Figure 1 illustrates the effect in a b-tree with 8 million initial index entries and insertion batches of 10,000 entries, with leaf splits per insertion batch varying from 0 to over 600. The details of the experiment are given in the experimental section. In order to reduce these wave of splits, alternative strategies for the initial bulk

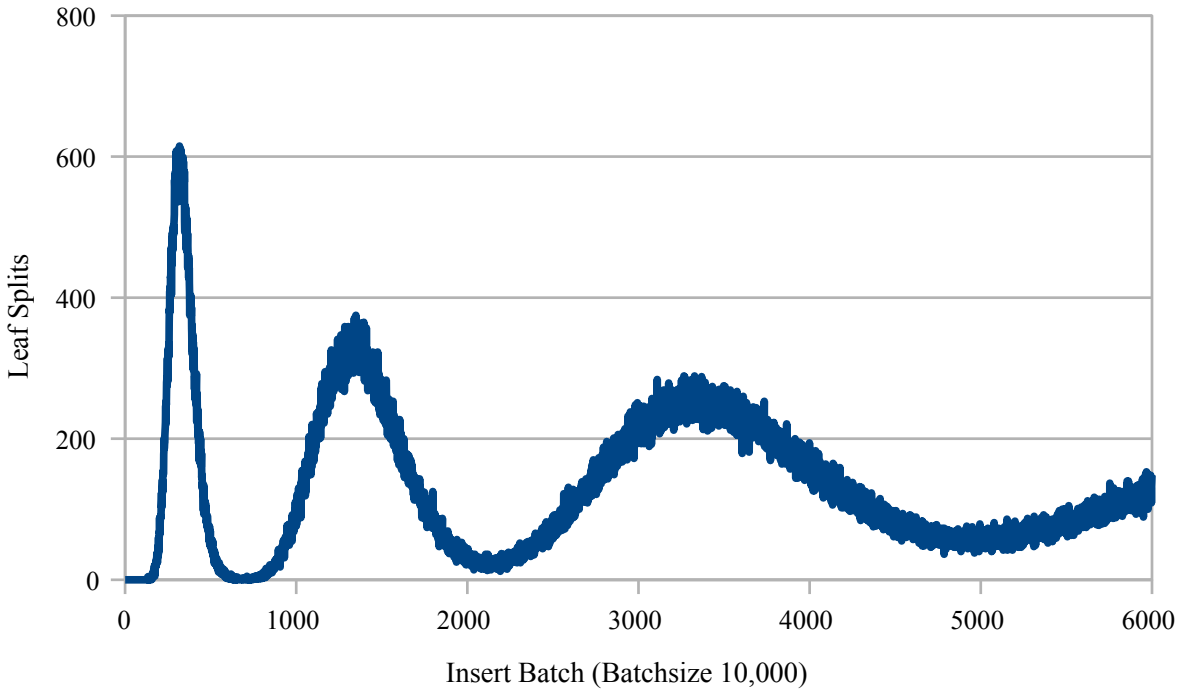

Fig. 1: Waves of Misery

loading of b-trees are needed. The main contributions of this paper are a theoretical analysis and solution for reducing splits as well as a variety of practical solutions that can easily be integrated into existing bulk-loading techniques.

In the remainder of this paper, the next section reviews related prior work. After a section illuminating the extent of the problem, two sections suggest both theoretically sound remedies and simple practical remedies or approaches to the problem. An evaluation section is then followed by a summary and suggestions for future research.

2 Related prior work

This section reviews relevant aspects of the well-known b-tree index structure. Knowledgeable readers may feel free to skip ahead.

2.1 B-trees

B-tree indexes are ubiquitous in databases, key-value stores, file systems, and information retrieval. There are many variants and optimizations [Gr11]; we review only those most

relevant to the new techniques proposed below. Suffice it to say that b-tree keys can be multiple columns (compound indexes), hash values (ordered hash indexes enable efficient creation, effective interpolation search, and easy phantom protection), space-filling curves (spatial indexes for multi-dimensional data and query predicates), or heterogeneous keys (merged indexes or master-detail clustering); and that both keys and values can be compressed, e.g., as bit vector filters instead of a set of row identifiers. B-tree creation benefits from sorting but sorting can also benefit from b-trees: runs in external merge sort in the form of b-trees, or even all runs within a single partitioned b-tree [Gr03] or a linear partitioned b-tree, enable effective read-ahead during a merge as well as query processing while a merge step is still incomplete. This is the foundation of log-structured merge-trees, discussed later. Sorted data permit efficient binary search; b-trees cache in their root node the first few keys inspected in any binary search (or reasonably good representatives for those key values), in the root's immediate children the next few keys inspected in a binary search, etc. In order to maximize the caching effect by minimizing key sizes, Bayer and Unterauer suggested suffix truncation when splitting a leaf node: rather than split rigidly into halves, i.e., at the 50% key, choose the shortest possible key value separating, say, the 40% and 60% keys [BU77]. Similar split techniques are also known from UB-trees [Ra00] and bulk-loading of R-trees [ASW12].

In order to avoid node splits immediately after index creation, e.g., during the first insertion into a new index, many implementation leave free space within each b-tree node. For a pretty typical example, Microsoft SQL Server supports two free space parameters during creation and reorganization of a b-tree index. The first one controls how much free space is left within each b-tree node, both leaf nodes and branch nodes (except along the right edge of the b-tree). The second parameter controls the number of empty pages, or more specifically the percentage of empty pages in the sequence of b-tree pages. These empty pages will be allocated during node splits. Leaving empty pages in this way ensures fast scans on traditional hard disk drives even after many insertions and node splits. SQL Server does not support free space fractions different for leaf and branch pages or free space fractions per key range.

In addition to the 'fill factor' option, the 'create index' statement in Microsoft SQL Server includes the option 'pad index,' which specifies whether to apply the fill factor not only to leaf nodes but also to branch nodes. Sybase supports the 'fill factor' option and a related option 'max rows per page.' The Oracle database and IBM DB2 support an index option 'percent free', which is the complement to the fill factor in SQL Server. The Oracle database also support many related options regarding the number of pages reserved for future index growth. Many other database systems support the Oracle or IBM syntax for free space within indexes. The aspect important in the present context is that all products support a fixed amount of free space in all index leaves, which causes waves of node splits.

2.2 Uneven page utilization and fringe analysis

Instead of using a fixed amount of free space, more flexible rules for the initial assignment of records to pages are required to mitigate or even avoid the waves of misery. A theoretically sound rule presented in this paper dates back to an analytical framework called fringe analysis [Ya78] that is originally used to prove the expected storage utilization of b-trees and other search trees. The framework of fringe analysis has been elaborated in [Ei82] and later applied to the analysis of various versions of b-trees [BL89]. However, none of the previous studies addressed the problem of bulk loading and the waves of misery identified in our work. Closely related to fringe analysis is spiral hashing [Ma79] where the utilization of the pages follows a logarithmic pattern. This avoids the undesirable oscillating search performance of linear hashing [La88] because the page with the highest expected load is always split next. However, the goal of spiral hashing is to guarantee a constant expected search performance while our approach addresses the problems of constant insertion performance and buffer contention.

2.3 Write-optimized b-trees

Traditional b-trees incur a high write penalty: modifying a single byte, field, or record forces a random page write to persistent storage. Write-optimized b-trees [Gr04] attempt to reduce this cost by turning the random write into a sequential write. More specifically, they employ append-only storage at page granularity, tracking new page storage locations within the b-tree structure, i.e., each page movement requires a pointer update in a parent page. Thus, write-optimized b-trees encompass the most crucial function of a flash translation layer (FTL). They do not permit neighbor pointers among sibling nodes; nonetheless, they permit comprehensive consistency checks (online/continuous or offline/utility) by fence keys in each page, i.e., copies of branch keys in ancestor pages. Write-optimized b-trees can suffer from poor scan performance on storage device with high access latency (e.g., seek and rotation delays). A possible optimization divides the key range into segments, stores each segment in continuous storage, and recycles replaced pages within a segment. The key range per segment may be dynamic, just as the key range per leaf node is dynamic in a traditional b-tree. The resulting design combines elements and advantages of write-optimized b-trees and of O'Neil's SB-trees [ON92]. On flash storage, segments may coincide with an erasure block.

2.4 Log-structured merge forests

Write-optimized b-trees optimize I/O patterns but still employ update-in-place within b-tree nodes (database pages). Log-structured merge-trees [ON96] employ random access only within memory but write to persistent storage only in append-only sequential patterns, each

page containing only new records. Thus, it seems that log-structured merge-trees solve the problem of write amplification. Log-structured merge indexes turn updates into insertions of replacement records and deletions into insertions of "tombstone" or "anti-matter" records. With only insertions remaining, the principal algorithm is that of an external merge sort, with runs in b-tree format and with the merge pattern optimized for both sorting and querying. On one hand, a high fan-in and few merge levels permit efficient sorting; on the other hand, as queries must search each one in the set (forest) of b-trees, efficient query processing demands eager merging and thus a low fan-in in each merge step. Bit vector filtering may reduce the number of b-trees a specific query needs to search, but merging in log-structured merge trees is generally less efficient than in external merge sort. In fact, small and sub-optimal merge fan-insertion can multiply the number of merge levels and thus the amount of I/O compared to an optimal merge sort. A second principal weakness of existing designs and implementations is that log-structured merge forests induce bursts of system activity in the form of merging. In other words, they don't avoid "waves of misery" but merely replace one kind with another [Gr19].

2.5 Summary of related prior work

Among traditional b-trees, write-optimized b-trees, and log-structured merge-trees, traditional b-trees and write-optimized b-trees offer the best query latency (single partition search), traditional b-trees and log-structured merge-trees the best storage utilization, write-optimized b-trees and log-structured merge-trees the highest write bandwidth (sequential writes only), and log-structured merge-trees the highest insertion bandwidth. Traditional b-trees and write-optimized b-trees suffer from the waves of misery addressed here; log-structured merge-trees have different waves of misery.

3 Problem assessment

While the introduction describes the problem in general terms and illustrates that the problem indeed exists, the present section illustrates when the problem occurs, when it does not, what characteristics or key value distributions cause the problem, etc. The following sections offer a deeper understanding as well as one particular solution; the following section offers more practical approaches and solutions. For b-tree leaf splits to occur in substantial frequency, the index (and its underlying table or data collection) must grow. A b-tree with little update activity and with little growth do not exhibit the issues discussed here. Little or moderate update activity (without overall growth) can be absorbed without splits if the initial index creation left free space in each node, as discussed in the preceding section. On the other hand, most databases have at least some tables and indexes that capture continuous business activity, whether those are banking transactions, web activity, sensor readings from internet-of-things devices, or other types of events. For b-tree leaf splits to occur in discernable waves, the key value distributions in the initial b-tree contents and in the

set of insertions must match. The higher their correlation, the sharper the waves. It is not required that this distribution be uniform or any other particular form. Figure 2 illustrates these points. If both initial key value distribution and insertions follow a normal or uniform distribution, there are fairly sharp waves of node splits that even overlap in Figure 2. If, on the other hand, their distributions differ, the waves are much less distinct. In a b-tree on

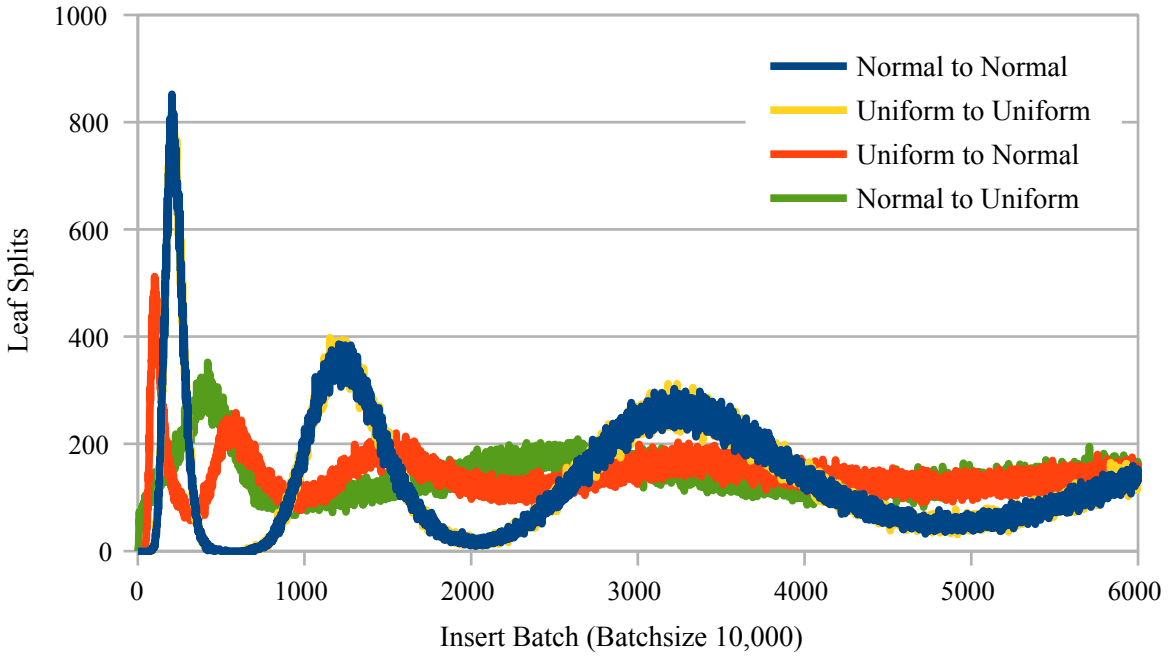

Fig. 2: Development of leaf splits for varying key distributions

hash values, a high correlation between these distributions is extremely likely. B-trees on hash values offer several advantages (over other traditional hash indexes), e.g., efficient creation after sorting future index entries, simple implementation of phantom protection (serializable concurrency control) by locking gaps between existing index entries, efficient merge joins after index scans, and more. Note that b-tree nodes with near-uniform key values permit interpolation search (instead of binary search) and that a b-tree root and its immediate descendent nodes can be cached as a single super-large node in memory. Thus, b-trees on hash values offer advantages but suffer from waves of leaf splits just as much as other b-trees, perhaps even more so because hash functions are specifically designed to produce uniform distributions and thus equal distributions in initial contents and insertions.

While free space left in each new b-tree node during index creation can absorb moderate update activity, it cannot absorb long-term growth. Initial free space merely delays the first wave of node splits, and it perhaps widens and weakens the first and subsequent waves, but it does not prevent them. Figure 3 highlights this point: even with as little as 50% initial space utilization (and thus also 50% free space on each index leaf), distinct waves occur. Thus, the facilities found in commercial relational database products and their commands for index creation do not prevent the waves of node splits addressed in the following sections.

The diagram in Figure 4 illustrates that index size matters. In a very small index (e.g., 1,000

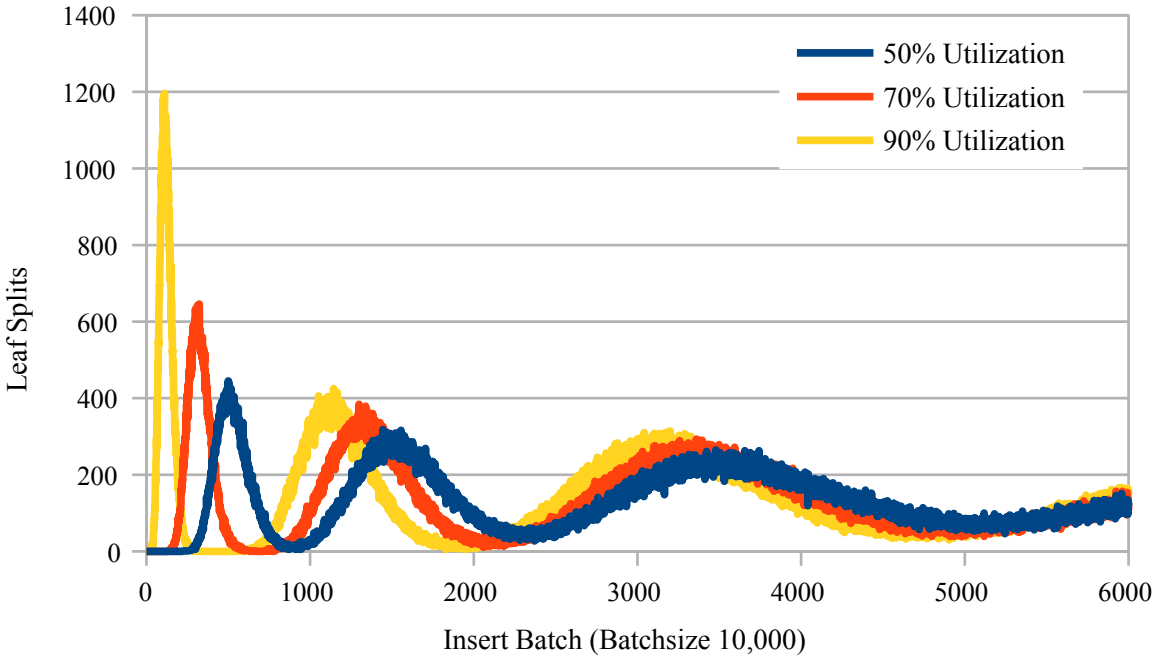

Fig. 3: Development of leaf splits for varying initial page utilizations

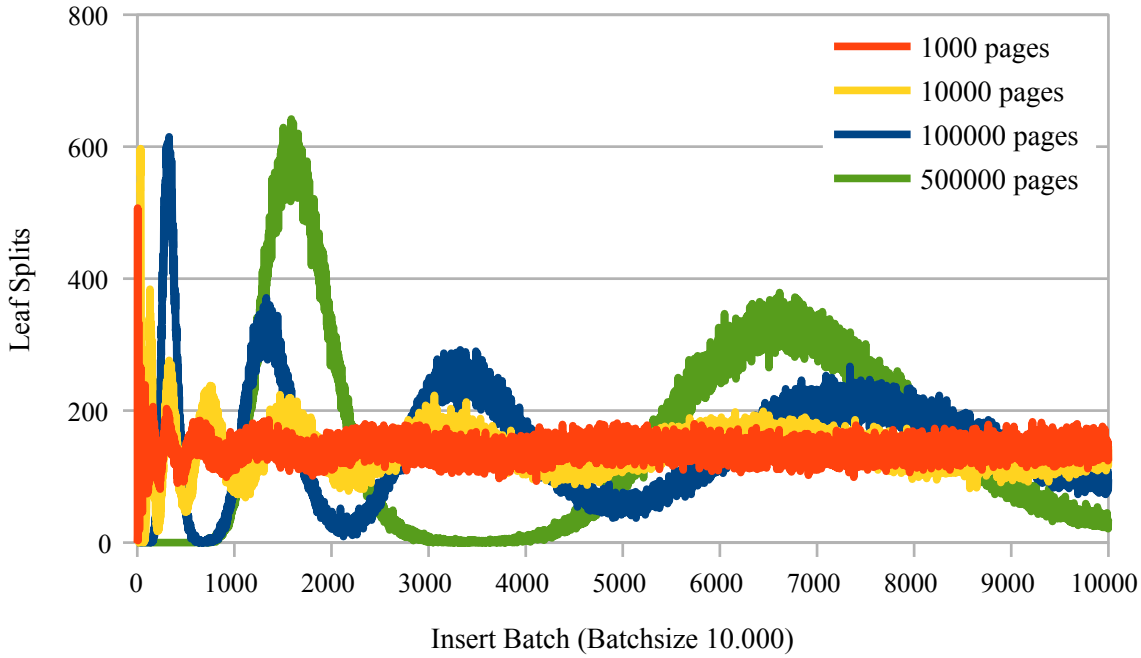

Fig. 4: Development of leaf splits for varying initial dataset sizes

leaf nodes), batches of 10,000 insertions force waves of node splits in rapid succession. They widen and weaken after a few waves. In contrast, in very large index (e.g., 500,000 leaf nodes), many batches are required to fill the initial free space in each node, but then there are distinct, high, and wide waves of node splits impacting database components such as the buffer pool for a longer period of time.

4 Sound remedies

The goal of this section is to provide a theoretically sound solution for loading b-trees for which succeeding insertions will trigger a split with a constant probability. Thus, there are no waves of misery. We limit our discussion first to the leaf level and later extend it to the upper levels of the tree. We assume records of constant size such that every leaf has a capacity of B records. For the sake of readability (and because of space limitations) let B being odd. Thus, a split of an overflowing page results into two pages each with (B+1)/2 records. A record has a unique comparable key that provides the ordering required for the b-tree.

The basic idea of the following approach is to keep the filling degree of the leaf pages in balance. Given a b-tree with n records, let $fd_i(n)$ denote the number of pages with i records, $(B+1)/2 \leq i \leq B$. Then, the probability that the next insertion triggers a split is $B * fd_n(B)/(n+1)$. As already shown in the introduction, this probability provides an oscillating behavior, our waves of misery, in case of extreme differences among $fd_i(n)$. It is also easy to see that the other extreme of a uniform fill degree with $fd_i(n) = fd_j(n)$, $(B+1)/2 \leq i, j \leq B$ and $i \neq j$, is not the theoretically sound solution because there are too many full pages and too less pages with a low utilization. The sound solution is somewhere between these two extremes such that the number of less populated pages is higher than the one of more populated ones.

A key idea of our algorithm is to employ the steady-state probability p_j that a page with j records occurs in the sound solution, $(B+1)/2 \leq j \leq B$. In an iterative manner, the algorithm randomly determines j with probability p_j, $(B+1)/2 \leq j \leq B$, and assigns the next j records from the input to a new page. The iteration stops when the number of remaining records is less than $(3B+1)/2$. Then the algorithm performs in the following way. If the remaining number of records is at most B, these records are stored in one leaf. Otherwise, two pages are created over which the records are equally distributed. Overall, this algorithm guarantees that all pages are at least half full as required for a traditional b-tree.

In order to analyze the split occurrence of the b-tree after loading, we assume equal distributions such that the distribution of insertions mirrors the key value distributions of the records in the b-tree. More formally, a b-tree with n records partitions the key domain into $n+1$ empty intervals and the probability an insertion hitting such an empty interval is

$1/(n+1)$. Note that this model is entirely different from the restrictive uniform distribution assumption.

Theorem 1 *Let us consider a b-tree with N records being created by our loading algorithm such that the probability q_j of the next insertion hitting a page with j records is $\frac{1}{\alpha(j+1)}$. Here, $\alpha = H_B - H_{(B+3)/2}$ denotes a weight parameter, where $H_j = \sum_{1 \leq i \leq j} 1/i$ is the partial harmonic series of size j. Furthermore, records are continuously inserted into the initial b-tree assuming equal distributions (i.e., insertion distribution mirrors the initial loading distribution). Then, the following two statements are fulfilled:*

- *First, the probability that a record insertion triggers a split of a leaf is constant, i.e., there are no waves of misery.*
- *Second, the probability p_j that a leaf consists of j records is given by $p_j = \frac{B \cdot \ln(2)}{j(j+1)}$.*

Sketch of the proof: The proof follows the basic ideas of fringe analysis [BL89; Ei82; Ya78]. In the following, we provide a rough summary and refer the interested reader to the original literature. For the following fringe analysis, we restrict our discussion to the leaf pages of the b-tree. The number of records partition the leaves into classes C_i, $(B+1)/2 \leq j \leq B$. Class C_i consists of all the pages with i records. For each class C_i, $f_i(n)$ denotes a counter for the corresponding number of leaf pages in a b-tree with n records. Let $q_i(n)$ denote the probability that an insertion will be in a page of class C_i. It follows that $q_i(n) = i \cdot f_i(n)/(n+1)$ since there are a total of $i \cdot f_i(n)$ records in pages of class C_i and every record is a left boundary of an empty interval where the next insertion may occur with probability $1/(n+1)$. Note that we make a negligible simplification because the first interval has not a record as its left boundary. All these probabilities are collected in a vector $\vec{q}(n)$. The basic idea of the fringe analysis is to keep track of this vector over a sequence of insertions. By using a quadratic transition matrix T with (B+1)/2 columns and rows it is possible to compute $\vec{q}(n+1)$ from $\vec{q}(n)$ in a recursive way ([Ei82]):

$$\vec{q}(n+1) = (I + \frac{1}{n+1}T) \times \vec{q}(n) \tag{1}$$

Here I denotes the identity matrix. The steady-state $\vec{q} = (q_{(B+1)2}, \ldots, q_B)$ of this recursive equation is then given by the solution of $T \times \vec{q} = \vec{0}$. In [Ei82], it is shown that this results in

$$q_j = 1/(j+1) \tag{2}$$

for $j = (B+1)/2, \ldots, B$. Because the leaf pages are filled in our initially loaded b-tree with N records such that the probability $\vec{q}(N+1) = \vec{q}$, it follows that $\vec{q}(N+k) = \vec{q}$ for $k = 1, 2, \ldots$. In particular, the probability $q_B(N+k) = q_B$ remains constant for $k = 1, 2, \ldots$ and so does the probability of a split. Thus, the first statement of Theorem 1 is fulfilled.
For the proof of the second statement, we make use of $q_j(n) \cdot (n+1)/j$ being the expected

number of buckets with j records [Ei82]. Moreover, $(n+1)/(B \cdot \ln 2)$ is the expected number of pages in a b-tree with n records because the expected storage utilization of a b-tree is $\ln 2$. The corresponding quotient of the two expected values is then $B \cdot \ln(2)/j(j+1)$. In fact, this quotient is asymptotically equal to p_j, the probability of a page comprising j records [BL89]. Thus, the second statement is also fulfilled, and our iterative loading algorithm is equipped with a theoretically sound rule for determining the filling degree of the next page.

So far, the discussion is limited to the leaf level only. Waves of misery still may occur for the index levels when a threshold-based loading approach is used. However, it is easy to see that our algorithm is also directly applicable to every index level. Also note that the algorithm runs online on an input stream, and therefore, the entire tree, including all index levels, is built from left to right as it is known from the standard loading algorithm of b-trees.

The probability vector introduced in Equation 2 is also known as the steady-state solution of the recursive equation. It is the foundation to show that the expected storage utilization is $\ln 2$ [Ya78]. Thus, the loading algorithm also creates b-trees with an expected storage utilization of $\ln 2$. This result can be guaranteed by changing the algorithm to a deterministic one where the ratio of pages with i records, $(B+1)/2 \leq i \leq B$ exactly matches the probabilities. Then, the loaded b-tree guarantees a storage utilization of $\ln 2$. Unfortunately, there is no steady-state solution with a higher storage utilization for the loaded b-tree. In order to obtain a higher storage utilization and a b-tree without waves of miseries, it is required to change the split policy of the b-tree by using partial expansions or elastic pages [BL89]. This is a direct extension of our approach and is not further elaborated in the paper.

The analysis in this section assumes that the key value distribution in the set of insertions mirrors that of initial b-tree contents. Moreover, it assumes that an overall space utilization of $\ln(2) \approx 69\%$ is sufficient for the application. Therefore, the following section offers some practical approaches that do not require these assumptions. The subsequent section offers an evaluation of both the techniques above and those below.

5 Practical remedies

With waves of nodes splits occurring in practice, although often not recognized, practical remedies are required that fit into scalable data center operations. For example, limiting initial or steady-state storage utilization to 69% is unacceptable. Therefore, many b-tree implementations attempt load balancing before splitting two nodes into three; and many database administrators frequently reorganize (rebuild) their indexes to reset free space per node to, say, 10%. Note that splitting 2-to-3 or even k to k+1 does not avoid waves of misery.

Even if the goal is to achieve, say, 10% free space in each node, it is not required to achieve 10% free space rigidly and precisely. Instead, the same overall objective is accomplished by leaving a different amount of free space in each node, chosen from a uniform random distribution from 0% to 20%, from 5% to 15%, or anything similar.

Another possible approach is to make this very systematic: 1st node 0% free space, 2nd node 1%, etc. to 21st node 20%; then restarting with 0% free space, etc. This approach favors some range queries and disfavors others, so an alternative assigns 0% free space to the 1st node, 20% to the 2nd node, 1% to the 3rd node, 19% to the 4th node, etc.

An entirely different approach assigns free space not numerically but by following the logic of suffix truncation (suffix compression) [BU77]. Originally conceived for splitting nodes in the middle, it can be adapted to index creation and reorganization when dealing with non-numeric keys. For example, instead of leaving precisely 10% free space, the key values at 80% and at 100% are compared, the shortest possible key value for the b-tree branch node determined, and the key values distributed to the current and next node according to this branch key. If there is no correlation between the shortest key value and its position, this approach promises a distribution of node utilization similar to the random approach above, with the compression effect added.

Finally, multiple of these techniques may be combined. For example, suffix truncation for the 1st node may look at the key values at the 80% and 100% positions, for the 2nd node at 70% and 90%, for the 3rd node at 79% and 99%, for the 4th node at 71% and 91%, etc. Other combinations may also be possible.

6 Evaluation

All experiments were conducted on a workstation equipped with an AMD Ryzen7 2700X CPU (8 cores, 16 threads) and 16GB of memory, running an Ubuntu Linux (18.04, kernel version 4.16). All of strategies were implemented using the Java indexing library XXL [Se01]. If not stated otherwise the b-tree was initially loaded using 100,000 pages of 8KB. The tree holds records consisting out of 21 integer values (totaling 84 bytes) of which 20 values are uniformly random. The last value is the key and is being randomly sampled according to a normal probability distribution. The loading phase is succeeded with 60 million record insertions. The results are shown based on the statistics of a batch of 10,000 insert operations. Having batches of 10,000 records is not uncommon and allows for a reduction of statistical noise. In the following, the practicality of the sound remedies, which are ideal for 69% utilization, are evaluated first. Afterwards, the various practical remedies also suitable for higher utilization requirements are analyzed.

6.1 Sound Remedies

To get a better understanding of the meaning of the sound remedies, the distribution of free space among all pages for an ideal solution (i.e., a b-tree in a steady state) and the proposed algorithm of the sound remedies section need to be evaluated. Naturally, in the ideal case, the overall space utilization amounts to $\ln(2) \approx 69\%$. Figure 5 depicts the results based on

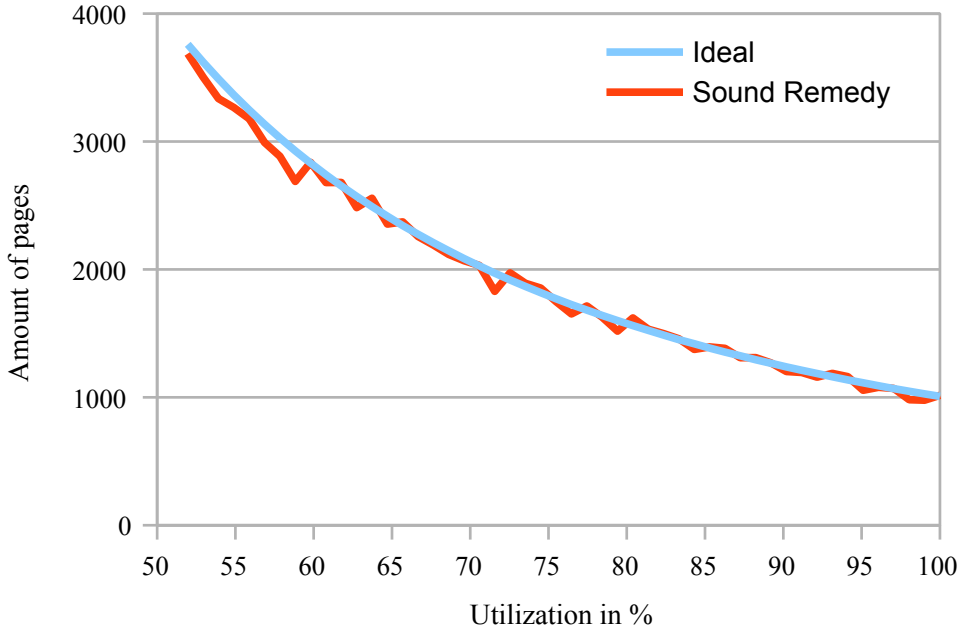

Fig. 5: Initial Page Distributions

the different utilization (x-axis) and the amount of pages having them (y-axis). The ideal solution shows the basic notion of the steady state: There are a few pages ready to be split right away (100% utilization, no free space), a lot pages not even near their split point (50%) and having all other utilization values in between them represented in the tree on a gradual slope. The sound remedy has the same features, but features some slight spikes in the slope. This is due to the the asymptotic nature of the algorithm and its inherent randomness - utilization is picked at random based on the described probability function. The latter is an important feature of the algorithm due to range queries: If the slope is perfect, a certain range of pages will have more free space than other pages. Thus, those ranges require more pages to be read.
As shown in Figure 6, the sound remedy also performs well in practice, if its assumptions are met. The x-axis represents the progression of insert batches while the y-axis shows the amount of leaf splits measured during each batch. While the constant strategy of having each page having the same utilization shows clear waves of splits, the sound remedy begins in the steady state and never leaves it.

6.2 Practical Remedies

Based on Section 5, three loading strategies were implemented that can easily be adapted into most systems in order to reduce the waves of misery. The most important parameters are the amount of data items to load and a target number of pages along with the overall space utilization. The analysis for each of the strategies is presented in turn. First, results for the *linear strategy* that systematically assigns free space to each page are shown. Second,

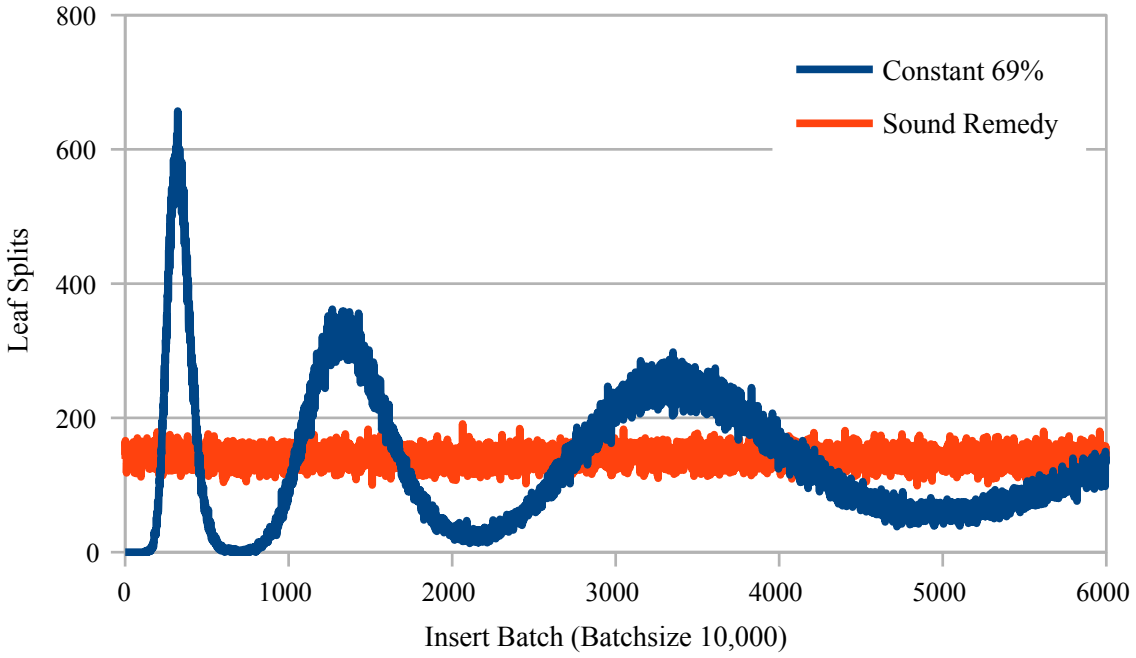

Fig. 6: Leaf Splits over time for ideal theoretical conditions (69% page utilization)

results for the *random strategy* that randomly assigns free space are presented. Third, a combination of both those strategies (*hybrid strategy*) is evaluated. Afterwards, the impact waves of misery can have on the buffer pool are discussed. The final subsection features the results on range query processing. As with the sound remedies, the constant strategy of uniformly loading each page with the same space utilization serves as a baseline for each evaluation.

6.2.1 Linear Strategy

The linear strategy assigns pages according to a linear function starting at 100% page utilization. The gradient and lower end of the function is computed based on the desired utilization and the amount of pages. In order to support range queries, values from the higher and lower end of the function are picked in an alternating fashion until both ends meet. At this point, all data has been successfully loaded into the tree. Figure 7 compares the impact of insertion of various initial utilization (70%, 80% and 90%) for the linear strategy. The y-axis shows the leaf splits while the x-axis indicates the total amount of batches inserted into the b-tree. Even at a high utilization of 90%, the linear strategy showcases slightly better leaf split performance than the constant strategy at 70%. Naturally, this means that achieving a similar split behavior requires less overall storage space for the same amount of data. Furthermore, the lower the utilization, the more the impact of waves is reduced. Even at 80%, the behaviour is somewhat close to the steady state and there are only short bursts of splits present.

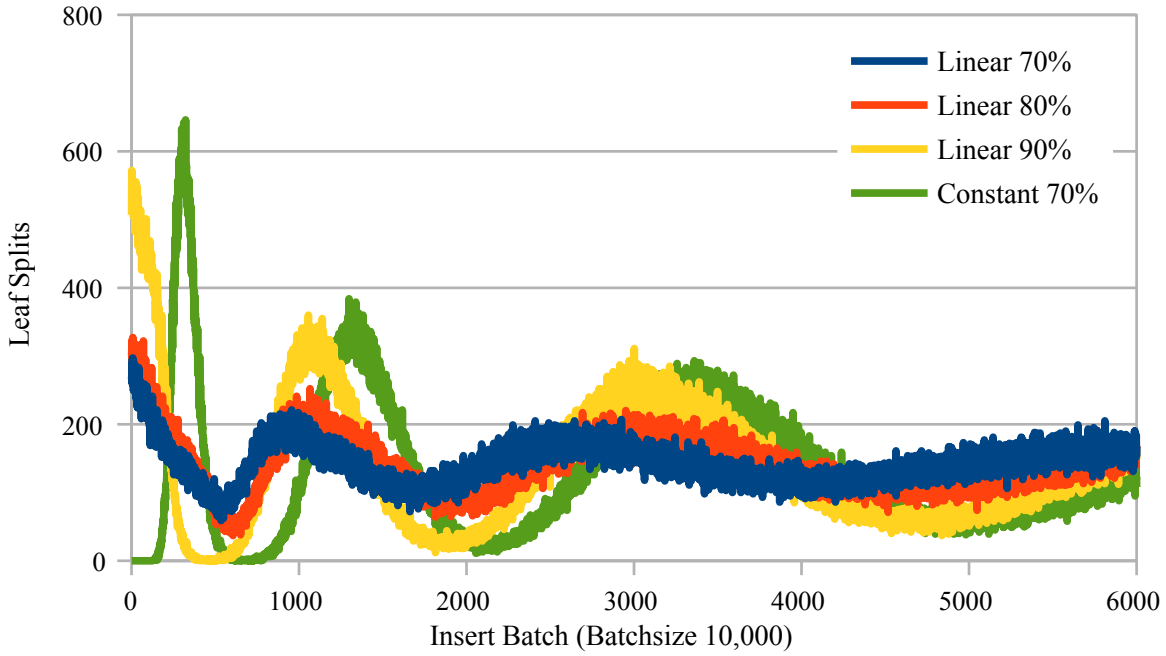

Fig. 7: Leaf Splits over time for linear strategies

6.2.2 Random Strategy

The random strategy chooses the page utilization of each successively loaded page randomly around the range of the given overall desired storage utilization. Due to randomness, the final page count may differ from the expected page count given the desired utilization, but those differences are expected to be negligible. Results presented in Figure 8 are based on a desired utilization of 80% with the same axis labeling as in Figure 7. The different lines present a varying range from which the random function chooses, i.e., *Random 5% Range* picks values from about 75% to 85% utilization while *Random 20% Range* picks values from 100% to 60%. A small range of 5% shows surprisingly little benefit compared to the constant strategy. However, wider ranges improve the desired effect significantly and tamper the waves of misery.

6.2.3 Hybrid Strategy

Both the linear strategy and the random strategy can significantly reduce the waves of misery and thus offer a good solution to implement in most systems. However, the strategies still show some slight waves when compared to the sound remedies at 69%. This is due to their initial load distribution. Similar to Figure 8, the results in Figure 9 showcases the page utilization distributions for the random and linear strategies in comparison to the ideal at 69% overall utilization. The utilization is on the x-axis and the amount of pages having it on the y-axis. The best result for the random strategy requires a random range of 31%, resulting in pages with less than 50% utilization. On the flip side, a lower range of 10%

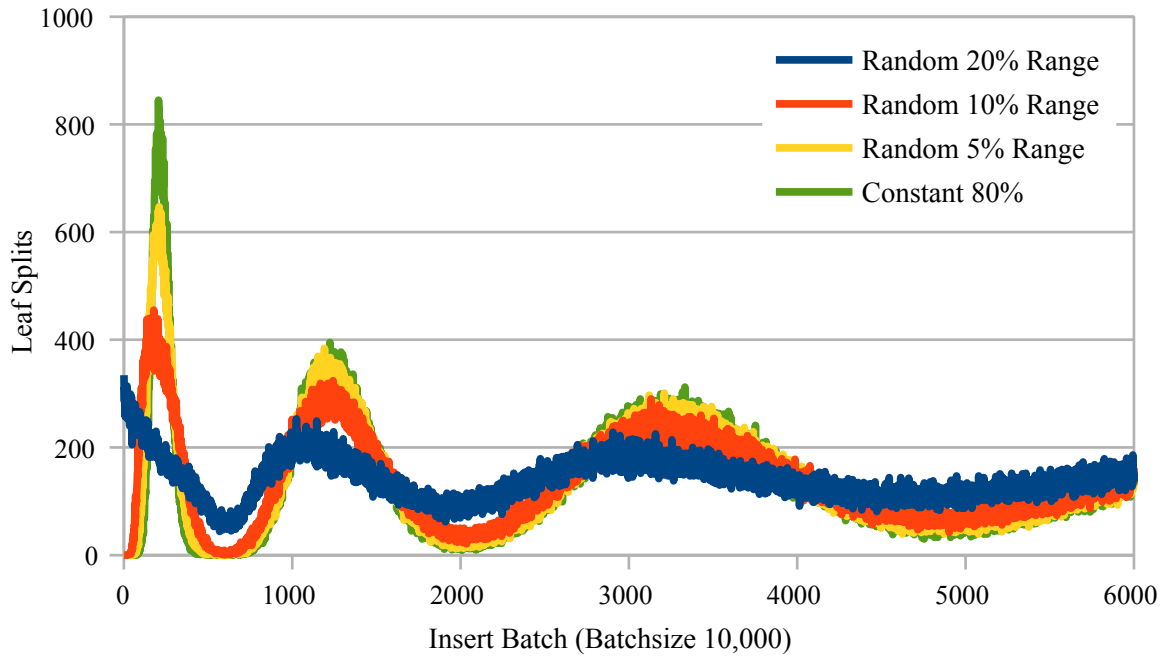

Fig. 8: Leaf Splits over time for random strategies

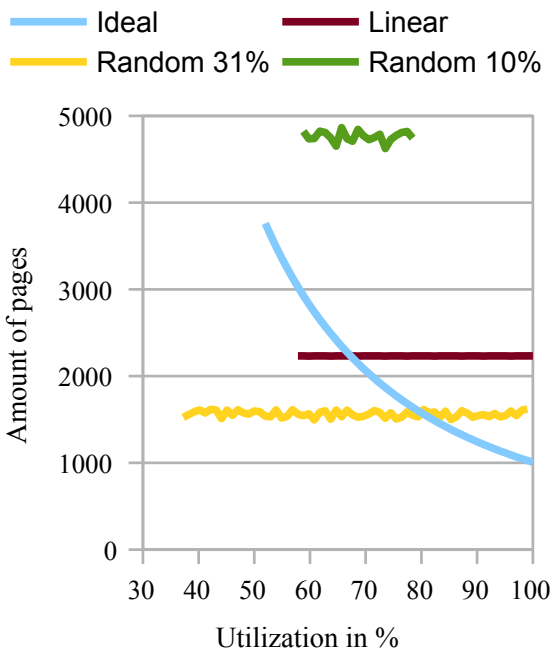

Fig. 9: Comparing the ideal initial page distribution with linear and random strategies

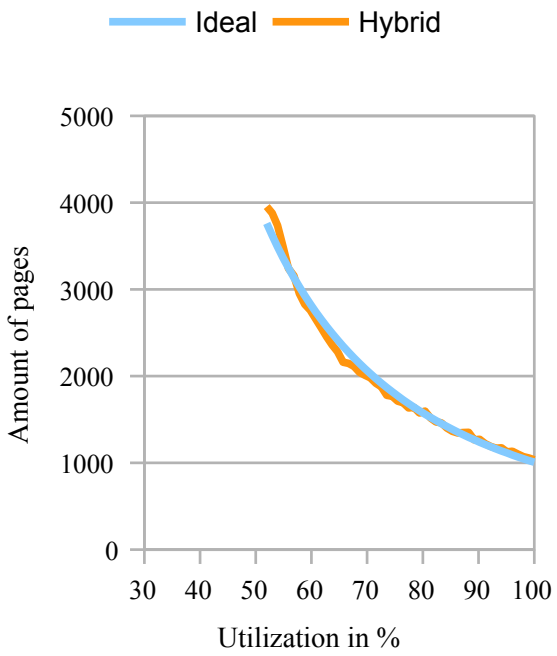

Fig. 10: Comparing the ideal initial page distribution with the hybrid strategy

does not lead to underful pages, but does not feature the overall range of different page utilization required for steady split rates. While the linear strategy features the right range of page utilization, their distribution is constant and thus not ideal.
To mitigate these deficiencies, a hybrid strategy which combines both ideas from linear and random was implemented. In order to reduce the rigid constant distribution of the linear strategy while keeping its overall range, only half of the pages are loaded according to this strategy. For the other half, randomness is applied as follows: Utilization is first randomly chosen between 50% and 100%. Afterwards, this range is narrowed linearly (i.e., lesser filled pages become more likely) until the bulk loading is complete. The resulting initial page utilization distribution is depicted in Figure 10. Clearly, combining those multiple simple strategies leads to a good approximation of the ideal solution.

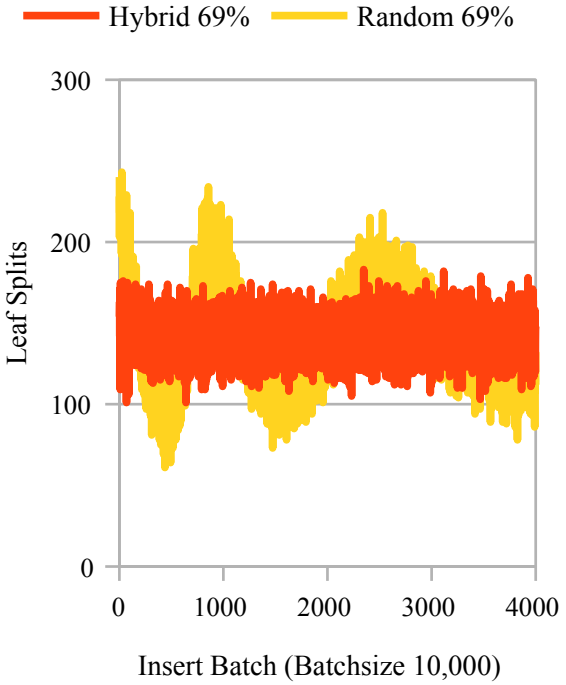

Fig. 11: Leaf Splits over time for hybrid strategy at 69% page utilization

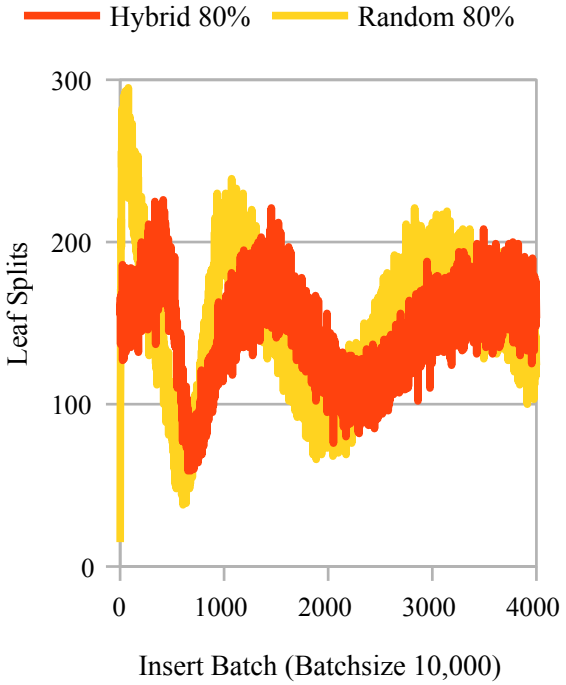

Fig. 12: Leaf Splits over time for hybrid strategy at 80% page utilization

Figure 11 verifies this statement by showcasing results for the insertion experiment for the first 4,000 insert batches at an overall target page utilization of 69%. The random strategy using a range of 31%, the best candidate for its class in terms of splits, is used as a point of comparison. The hybrid strategy outperforms it and is in the steady state from the get-go. The same experiment was repeated for 80% utilization (Figure 12). Since the b-tree does not have the theoretically ideal target utilization, the hybrid strategy does suffer from slight waves. However, it is still able to outperform the random strategy and overall has significantly reduced the impact of splits. Furthermore, while this experiment focuses on numeric keys, the hybrid strategy can be adjusted for non-numeric keys by using suffix truncation instead of randomly choosing keys within the ranges.

6.2.4 Buffer Pool Utilization

Up until now, the evaluation focused on split frequencies, which are a natural indicator of I/Os and buffer pool contentions. To showcase the effects on the actual buffer, the experiments were verified by deploying an LRU buffer and measuring its statistics. To analyze leaf splits separately, separate buffers for inner and leaf nodes were used.
The first experiment utilizes a buffer with 10,000 pages and compares the sound remedy strategy with a constant strategy with a target utilization of 69%. Figure 13 showcases

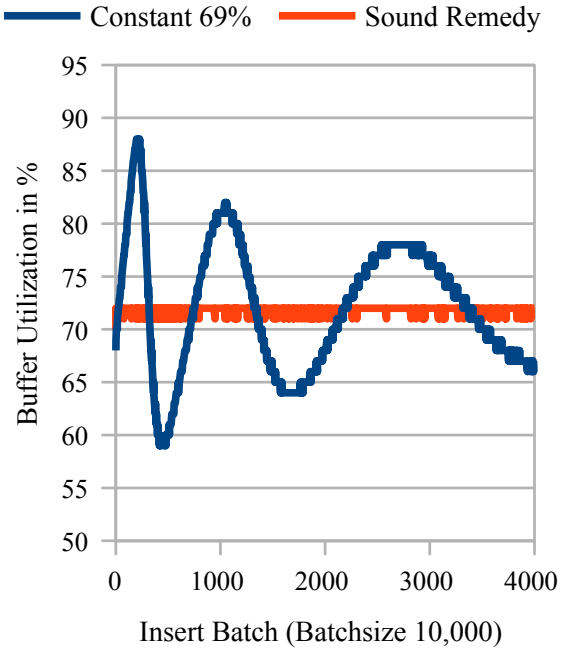

Fig. 13: Buffer pool utilization over time for 69% page utilization

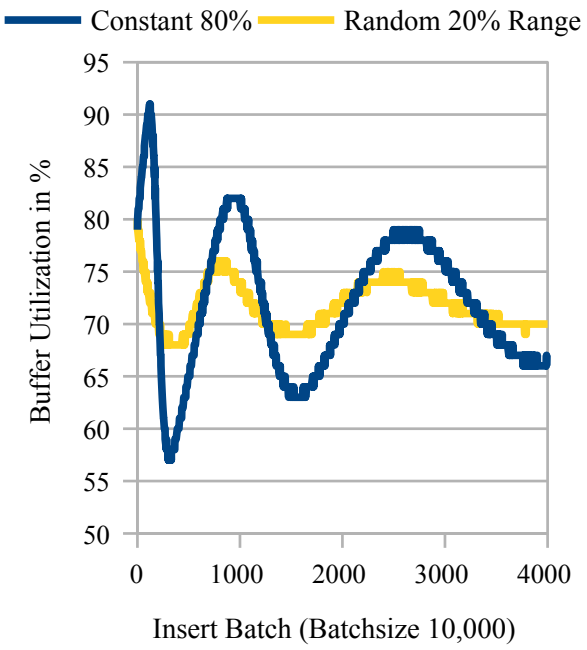

Fig. 14: Buffer pool utilization over time for 80% page utilization

the results. Here, the y-axis shows the buffer utilization, i.e. the quotient of used buffer pool bytes to totally available raw bytes. Clearly, the waves of misery also appear in the buffer utilization for the constant strategy while the algorithm based on the sound remedies eliminates them. The experiment was repeated for a higher overall utilization of 80%. This time, the constant strategy is compared to a random strategy featuring a range of 20%. As Figure 14 shows, waves for the constant strategy are even higher than at 69%. Furthermore, the random strategy reduces them just as it reduced the amount of leaf splits.

Finally, both constant and random strategies at 80% overall space utilization were compared for different buffer sizes based on buffer evictions, i.e. leaf pages being written out from the buffer onto disk due to other reads or writes requesting another page. Figure 15 depicts the total amount of evictions (y-axis) per batch (x-axis) while having varying buffer sizes (10,000, 20,000 and 100,000 pages) for each strategy. At 10,000 and 20,000 maximum buffer capacity, the waves of misery in form of increased spike of write operations are clearly visible for the constant strategy. Meanwhile, the random strategy almost eliminates the effects. For a larger buffer size of 100,000 pages, the additional writes are obviously

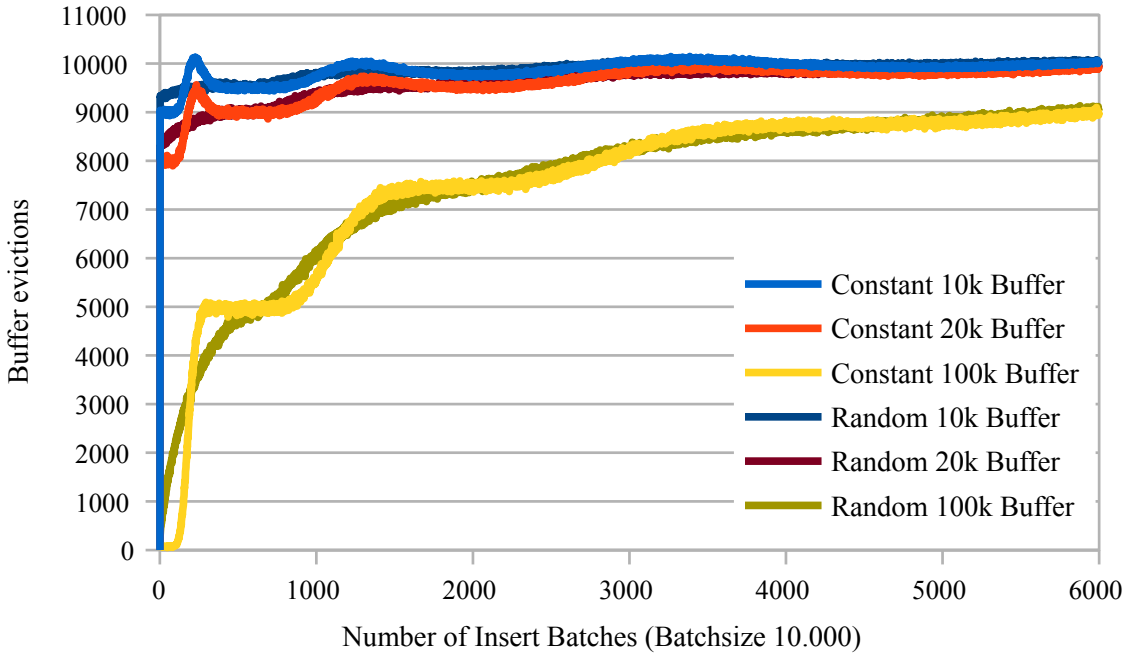

Fig. 15: Buffer evictions (writes) over time for varying buffer sizes

absorbed into buffer pool, causing only buffer contention and additional split operations, but no spikes in I/Os. Nevertheless, since the b-tree increases in size, more pages have to be constantly written out. The random strategy allows for a more gradual increase. Meanwhile, the constant strategy shows sharper *steps* at the points of the waves of misery since there is a sudden increase in new pages.

6.2.5 Range Queries

Since the proposed strategies do not load pages with a uniform free space, this obviously has some impact on range queries. For example, in the simple case of uniform key distribution, there is a chance that the same range span may hit varying degrees of full pages for different key ranges in the key space, resulting in more page reads and a negative performance impact.

To analyze this impact, the whole key space of a tree was queried using a succession of jumping range queries. First, a tree was loaded with 500,000 pages at a targeted utilization of 80% using an uniform key distribution ranging from the integer key 0 to the integer key of 100 million. Afterwards the range is successively queried with key ranges spanning 100,000 values starting from the first key, then querying the next 100,000 values, etc. until the last key is reached. Figure 16 features the results of this experiment for variations of the linear strategy. The starting value of the range query is depicted on the x-axis and the number of (leaf) pages accessed in the query on the y-axis. The *strict linear* strategy refers to a linear distribution which does not alternate between page sizes. It performs best for the first couple of ranges, because those ranges have the most full pages. On the flip side, it

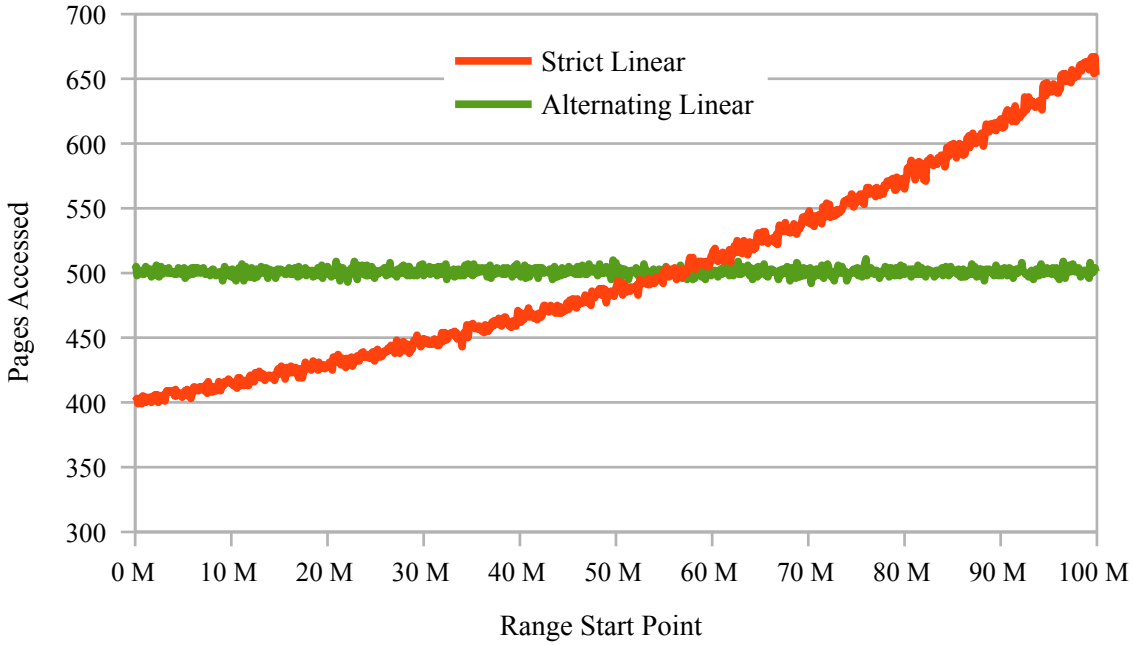

Fig. 16: B-tree range queries (range 100k) with varying starting points that cover the whole tree range

deteriorates for last ranges, since those ranges feature less occupied pages. The *alternating linear* strategy reduces this variation in performance and constantly accesses 500 pages for each read operations. We also repeated this experiment for the constant strategy, which performs about the same as alternating linear strategy. Furthermore, the random strategy also performs in the same ballpark, but, as expected, has slightly more spikes due to its randomness.

7 Summary and suggestions for future work

In summary, research in the past has overlooked the waves of node splits starting soon after b-tree creation, load, or reorganization. There are multiple means for avoiding or reducing them; many of them are both simple and effective. Among them, a free space target plus suffix truncation during leaf splits (as recommended decades ago in the context of prefix b-trees) offer the best combination of multiple advantages. Put differently, while suffix truncation during leaf splits has always been a good idea for minimizing the number of branch nodes and for speeding up root-to-leaf traversals, it is now clear that it is also a good idea and an effective means for avoiding multiple waves of misery after each index creation, load operation, and reorganization. Ongoing research focuses on “waves of misery” in log-structured b-tree forests [Gr19]. Future work will further research “waves of misery” on b-tree branch nodes, waves of node splits for specific key calculations such as space-filling curves, their forms and impacts in index and storage structures other than b-trees, specific forms of waves after specific load sequences such as sorting before loading R-trees, and what remedies may apply in the context of those index and storage structures.

Acknowledgments

This work has been partially supported by the German Research Foundation (DFG) under grant no. SE 553/9-1.
The idea for this work was made possible through discussions held at the Dagstuhl seminars 17222 (“Robust performance in database query") and 18251 (“Database architectures for modern hardware").

References

[ASW12] Achakeev, D.; Seeger, B.; Widmayer, P.: Sort-based query-adaptive loading of R-trees. In: CIKM'12. Pp. 2080–2084, 2012.

[BL89] Baeza-Yates, R. A.; Larson, P.: Performance of B+-Trees with Partial Expansions. IEEE Trans. on Knowledge and Data Engineering 1/2, pp. 248–257, 1989.

[BU77] Bayer, R.; Unterauer, K.: Prefix B-Trees. ACM Trans. on Database Systems (TODS) 2/1, pp. 11–26, 1977.

[Ei82] Eisenbarth, B.; Ziviani, N.; Gonnet, G. H.; Mehlhorn, K.; Wood, D.: The Theory of Fringe Analysis and Its Application to 2-3 Trees and B-Trees. Information and Control 55/1-3, pp. 125–174, 1982.

[Gr03] Graefe, G.: Sorting And Indexing With Partitioned B-Trees. In: CIDR. 2003.

[Gr04] Graefe, G.: Write-Optimized B-Trees. In: VLDB 2004. Pp. 672–683, 2004.

[Gr11] Graefe, G.: Modern B-Tree Techniques. Foundations and Trends in Databases 3/4, pp. 203–402, 2011.

[Gr19] Graefe, G.: Waves of misery in b-tree forests, in preparation, 2019.

[La88] Larson, P.: Dynamic Hash Tables. Communications of the ACM 31/4, pp. 446–457, 1988.

[Ma79] Martin, G.: Spiral storage: Incrementally augmentable hash addressed storage, Theory of Computation, tech. rep., University of Warwick, 1979.

[ON92] O'Neil, P. E.: The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. Acta Informatica 29/3, pp. 241–265, 1992.

[ON96] O'Neil, P. E.; Cheng, E.; Gawlick, D.; O'Neil, E. J.: The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica 33/4, pp. 351–385, 1996.

[Ra00] Ramsak, F.; Markl, V.; Fenk, R.; Zirkel, M.; Elhardt, K.; Bayer, R.: Integrating the UB-Tree into a Database System Kernel. In: VLDB 2000. Pp. 263–272, 2000.

[Se01] Seeger, B.: eXtensible and fleXible Library (XXL) for Java, 2001, URL: https://github.com/umr-dbs/xxl, visited on: 09/30/2018.

[Ya78] Yao, A. C.: On Random 2-3 Trees. Acta Informatica 9/2, pp. 159–170, 1978.