SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluating a Radix Tree Cache for Database Management Systems

Matteo Wohlrapp

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluating a Radix Tree Cache for Database Management Systems

Evaluierung von Radix Tree Caches für Datenbanksysteme

Author:	Matteo Wohlrapp
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Philipp Fent, M.Sc.
Submission Date:	15.08.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, Matteo Wohlrapp

Abstract

Traditionally, database management systems (DBMS) have stored data on disk and transferred it to the main memory for processing because of the limited primary storage capacity. However, with increasing main memory availability and affordability, in-memory DBMS are becoming a practical and faster alternative. Simultaneously, we must recognize the progressions in external storage technology and concede that main memory still costs significantly more than disk space. Therefore, it may be worth exploring the possibility of combining both concepts to exploit their individual strengths. We propose an approach that integrates in-memory and disk-based systems by deploying an adaptive radix tree (ART) based on the concept presented by Leis, Kemper, and Neumann as a caching mechanism for a traditional disk-based B+ tree. The system undergoes evaluation on a range of workloads spanning different operations. The findings suggest it performs more than twice as well for read, update, and scan operations. Nevertheless, the cache exhibits some limitations for insert and delete operations, introducing extra overhead. Additionally, our results have revealed that the indices' distribution significantly affects the system's performance. In particular, we noted that skewed loads improve the cache's performance, drawing attention to the effect of the specific use case on the effectiveness of our suggested system.

Contents

1. Introduction

Database Management Systems (DBMS) have become increasingly important in the information age. Today, almost no major software solution works without the support of a DBMS. Two types can be distinguished: in-memory and disk-based. In-memory systems store all information in the computer's main memory, while disk-based systems store data externally and load it into the main memory for processing. According to the memory hierarchy, accessing external storage introduces a significant performance overhead compared to accessing main memory [9].

In recent decades, hardware trends have had an important impact on the development of DBMS. Initially, the limited capacity of main memory meant only a fraction of data could be stored in it [14], leading to a dominance of disk-based systems [7]. Increasing amounts of data reinforced the trend, with many systems now commonly storing terabytes or even petabytes (10^{12} - 10^{15} bytes). Dedicated index structures, such as B+ trees, have been developed to organize storage beyond main memory, allowing efficient data access [8].

Despite these limitations, researchers have explored in-memory databases since the early days of computing. Originally, these systems were primarily used for performance-intensive applications due to low capacity and high memory prices [7]. With main memory being more readily available today, it can handle most use cases effectively. As a result, newer index structures such as the radix tree were introduced to manage in-memory systems more efficiently [13].

However, external storage access times have decreased dramatically along with the increased size of main memory, and modern SSDs can now read up to 3.5 GB per second. Since main memory is still relatively expensive compared to SSDs, combining large main memory with fast external storage can be an effective option [14]. This thesis presents a solution that couples a radix tree, used in in-memory applications, as a cache to a B+ tree, commonly used in disk-based systems. The performance of this combination is then analyzed and compared to a non-cached solution.

The thesis is structured as follows: Chapter Two introduces disk-based DBMS and defines the key concepts. In Chapter Three, we explain in-memory systems in more detail. Chapter Four then explores using a radix tree as a cache for disk-based DBMS, and Chapter Five evaluates the system's performance. Chapter Six finally presents the conclusion of the thesis.

2. Disk-Based Database Systems

This chapter provides an overview of disk-based database systems. These systems have been the subject of extensive research and development, and their practicality has been well-established [9]. The first part of the chapter describes the general concepts of storage management and the interaction between disk and main memory. Subsequently, we will introduce different abstraction levels for data representation. Furthermore, we present an index structure that enables efficient data access.

2.1. Memory Management

Database management systems are designed to handle large amounts of data. In the past, these systems required more capacity than the main memory could provide. To overcome this problem, disk-based systems offload data to external storage to persist it. However, accessing data on disks takes up to 10^5 ns longer than accessing main memory. The difference in access times can create a bottleneck, and disk-based storage systems have implemented various techniques to compensate for this [9]. Essential aspects of storage management include spatial and temporal control [10] and abstraction of the underlying hardware [9].

2.1.1. Files and Blocks

To interact with data objects, they must reside in main memory. A *storage manager* implements a file system to manage data and its transfer between the disk and main memory. This can either mean utilizing the functionality provided by the operating systems or managing the storage directly by maintaining a record of the location of files and blocks on the disk [8]. The implementation of the file system improves both spatial and temporal control abilities of the database management system, offering more control, which can lead to increased efficiency [10].

Data is generally stored in blocks that can be addressed by numbers indicating their physical location on external storage. Various concepts influence the performance and flexibility of disk management, depending on the system's requirements.

Static mapping allocates continuous memory on the disk and saves blocks sequentially. This results in a straightforward calculation of the offset for a block and can reduce access times because sequential bandwidth is significantly faster than random access [10]. However, a disadvantage of this approach is that the file must be fully allocated at the beginning of the run.

In contrast, dynamic mapping allows for memory allocation to take place dynamically. Single memory blocks are allocated if more memory is required and the reference is saved. Although this approach does not require assumptions about the size in advance, it scatters blocks all over the disk, which leads to decreased performance.

Dynamic extend mapping combines both of these ideas. It allocates successive memory areas that accommodate multiple blocks when additional space is required. These areas are referenced, and each file descriptor is linked to an extent table. The extent table tracks each block's number and starting address in the extension [9].

2.1.2. Segments and Pages

Segments and pages, representing a logical address space, introduce an additional abstraction layer. They serve a diverse range of dedicated functions, including storage for intermediate results, user data, or providing concurrent data access services. A segment S comprises several pages P_i of equal size L. It enables logical mapping to physical addresses [1]. Two distinct options are available to facilitate such mapping. Direct page mapping saves pages sequentially in a file by assigning the page P_i with $i \in [0, s]$ to reference block B_j with $j \in [k, k+s]$. Indirect page mapping introduces a table that maps pages to blocks scattered throughout the file. This approach provides greater flexibility, but it also results in increased access time due to additional indirection. While it is impossible to split segments over multiple files, storing multiple segments within a single file is possible. Furthermore, read or write access is only possible at the page level. Within a page, various concepts can manage individual entries containing complex types, tuples, clusters, or entries without a size limit [9].

2.1.3. Buffer Management

A *buffer manager* is introduced to efficiently access pages in main memory. The buffer manager controls a pool of pages loaded from the disk, combined with metadata as frames. Apart from spatial allocation, the buffer manager handles temporal aspects by adjusting the pool size according to system requirements. A hash table is used to implement the buffer pool, which maps page_ids, unique identifiers for each page, to their respective frame location and includes information about the memory position on the disk [9]. Figure 2.1 illustrates the concept. The frame consists of a flag indicating whether a page is dirty or pinned. This information becomes critical when writing pages to disk; the system must persist dirty pages first and cannot exchange pinned pages as they are still in use. Additionally, information for page eviction strategies is stored, which increases cache locality [10].

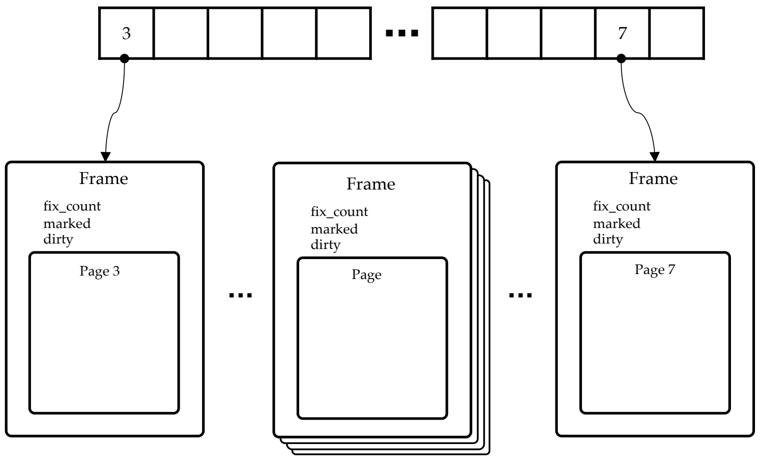

Figure 2.1.: Hash map of the buffer manager, mapping page_ids to frames

2.2. Index Structures

The efficient locating of records is crucial in DBMS due to the disk storage's high access time, which makes retrieving only relevant pages necessary. To minimize the number of pages accessed, index structures are used to increase the lookup speed. Data can be indexed sequentially, scattered, or held in a tree structure [9]. The following section will explain the B+ tree as a specific type of tree structure in more detail.

2.2.1. B+ Tree

The underlying principle of a B+ tree is similar to that of a regular B tree [3]. For an associated parameter k:

1. Every path from the root to a child has the same length, called height
2. Every inner node but the root and leaves have at least $k + 1$ children, where k is a parameter associated with the tree. The root is either a leaf or has at least two children
3. Every node has $2k + 1$ children at most

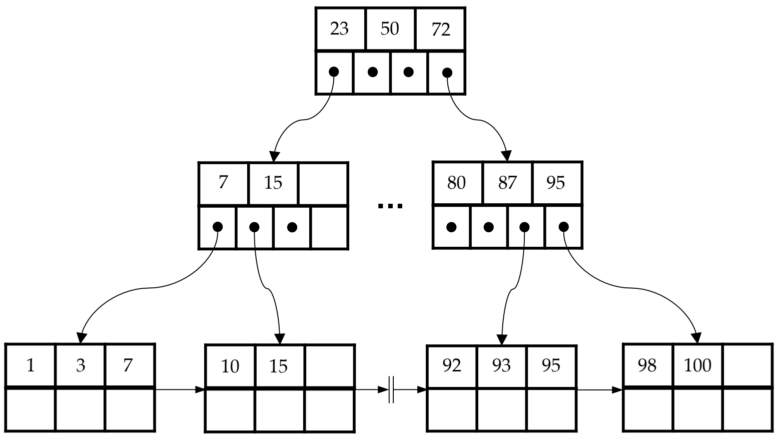

Figure 2.2.: Illustration of a B+ tree

A B+ tree consists of inner nodes and outer nodes or leaves. Unlike a regular B-Tree, a B+ tree stores all information in the leaves. An illustration of such a tree is shown in Figure 2.2.

The inner nodes comprise a set of ordered keys $k_i, i \in [0, k]$ and corresponding pointers $p_j, j \in [0, k+1]$ to child nodes. For every $i \in [0, k]$, it holds that each key in p_i is smaller or equal to k_i, while every key in p_{i+1} is bigger than k_i. The outer nodes also contain an ordered set of keys with corresponding values, where each key k_i is directly associated with a value v_i. The value can either be stored directly in the node, or it can be a pointer, introducing another level of indirection. In general, a B+ tree with a maximum of $2k+1$ children has a height between $1 + \log_{2k+1}(\frac{N}{2k})$ and $2 + \log_{k+1}(\frac{N}{2k})$ for $h \geq 2$ [9].

When searching for a value in the tree, we can locate the smallest value greater than the desired key. Consequently, we can follow the associated pointer recursively to the correct leaf node. If the lookup key is greater than every key in the node, we must follow the path to the rightmost child, as there is no corresponding key [8]. Maintaining the invariants when inserting or deleting is crucial to ensure the tree's functionality.

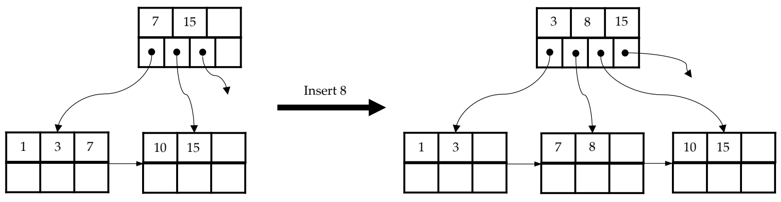

Figure 2.3.: Splitting a B+ node

To insert an item, we first identify a suitable leaf node. The key-value pair is inserted if the leaf node has enough space, maintaining the current elements' order. If there is not enough space in the leaf node to accommodate the new element, we split it into two separate nodes. This process requires creating a new node and transferring half of the key-value pairs. Subsequently, we update the parent node with a new key and a reference to the newly created node. This process is illustrated in Figure 2.3. If the parent node is also full, this procedure can have a cascading effect, requiring the repetition of the operation until enough space is available within a parent node. If an insertion into the root node is impossible, a new root node containing the two nodes is created by splitting the previous root node [8].

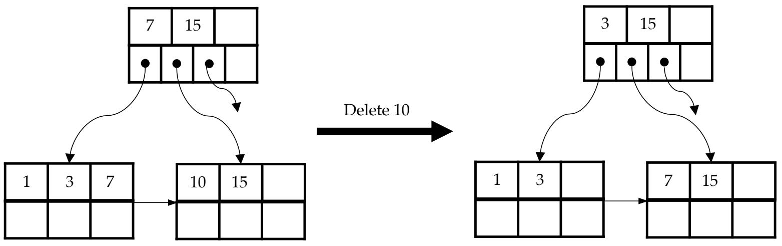

Figure 2.4.: Substitution in a B+ tree

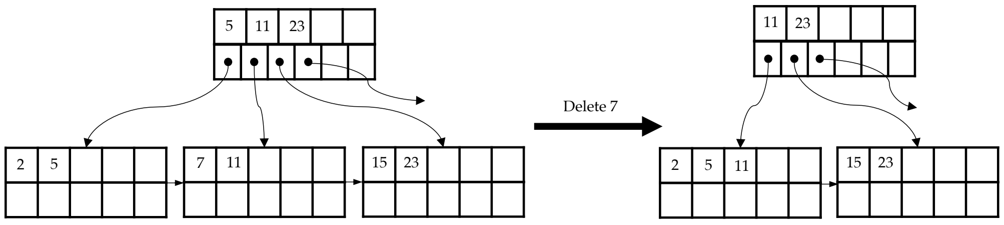

Figure 2.5.: Merging a B+ node

To delete an item, the initial step is to find the appropriate leaf node. Should an inner node containing the key to be deleted be found while traversing the tree, the key must be replaced with the next smaller key. After locating the correct leaf node, we can remove the key-value pair, respecting the order within the node if it contains more than the minimum number of elements. However, if the node has already reached the minimum, we must perform operations to prevent the number of entries from becoming too small. Two techniques, namely substitution and merging, are used for this purpose [8]:

- Substitution borrows an element from a neighboring node. If a neighboring node on the same level with more elements than the minimum required exists, we transfer a key from the parent node into the present node. A key from the neighboring node takes the place of the transferred key within the parent node. This procedure is depicted in Figure 2.4.

- Merging is only used when substitution is not possible. If the neighboring nodes on a similar level contain less or equal to the minimum number of elements, merging is performed, and the dividing key between the two nodes from the parent is placed in between, as shown in Figure 2.5. This action can cascade, causing repeated deletions in the parent nodes. If the root node only has one entry following the deletions, we replace it with its only child and later delete it from the tree.

3. In-Memory Database Systems

This chapter presents an overview of in-memory databases. Historically, these systems have not been the main focus of research in the field. Nevertheless, due to the improved main memory capacity, they have become increasingly relevant [7]. After reviewing past and recent developments, this chapter presents an index structure for in-memory databases.

3.1. Developments

Latency is lower when accessing main memory compared to disk storage. However, the limited availability of RAM meant that in-memory databases were first used for performance-critical applications in finance or telecommunications [7]. The first research was conducted in the 1980s, focusing on optimizing existing database systems through extensive use of main memory [5]. Further studies explored index structures and proposed solutions tailored explicitly for in-memory databases [12]. Since then, the cost of main memory has decreased by a factor of ten every five years [17]. This price decrease has encouraged research and led to the integration of in-memory solutions in several notable systems, including SAP HANA [17], Oracle Times Ten [11], and Hekaton [6].

3.2. Index Structures

With increasing research on in-memory DBMS, new methods of organizing and indexing data were developed to improve CPU efficiency [7]. The CPU now binds most traditional database systems due to processor and architectural changes [13]. This has inspired the creation of new index structures. The T-tree was the first example of such a paradigm shift, combining the binary search properties of the AVL tree with the storage capacity of the B+ tree [12]. However, its association with binary search trees made the structure inefficient, as hardware developments disrupted the assumption of constant memory access times [13]. Other approaches involved adjusting the node size in a cache-sensitive manner. However, this did not reduce the number of comparisons within the index structure [15] and thus did not tackle CPU stalls caused by unpredictable comparison results [13]. More recent research has introduced novel approaches to overcome this problem. The K-ary search tree [16] employs SIMD instructions to parallelize data-level comparisons during traversal. However, it depends on the linearization of the tree. This approach leverages modern architecture but entails high maintenance expenses due to the associated costs of preserving the linearized tree representation [16]. Besides trees, hash tables are another option that utilize the increasing main memory capacity. Hash tables provide access times of $O(1)$ for point queries, but they have limitations in handling growth and range queries [13]. The presented index structures offer a broad range of features using less efficient trees or fast point queries. The subsequent section elaborates on an approach that operates on key prefixes to achieve constant access while maintaining the order of elements.

3.2.1. Radix Tree

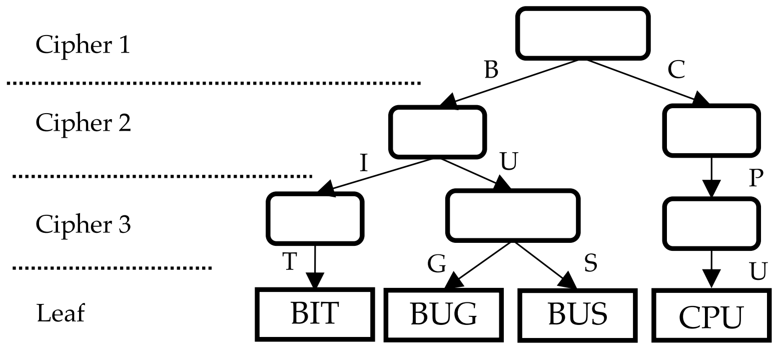

Figure 3.1.: Illustration of a radix tree without compression

A *radix tree* is an index structure based on a prefix tree or trie [2]. Every node in the tree represents a segmented part of the key. Each child node shares a common prefix with its parent, so the complete key is derived by combining the partitioned parts of the visited nodes. Figure 3.1 demonstrates that the full key is obtained by traversing the edges of the structure. In addition, radix trees store the keys in lexicographic order, and all insertion sequences result in the same tree [13]. In contrast to a standard trie, a radix tree is more memory-efficient, compressing child nodes whenever paths remain unused [2]. This is depicted in Figure 3.3, where a single node represents the whole prefix 'CPU'.

In the context of DMBS, unique keys of length k are inserted into the tree. As a result, the tree's height depends solely on the key's size rather than the number of elements in the structure. Therefore, all operations have a complexity of $O(k)$, which is particularly useful when the key size does not scale with the number of entries [13].

In this paper, 'radix tree' refers to the 'adaptive radix tree' (ART) developed by Leis, Kemper, and Neumann in [13]. We use the two terms interchangeably. The ART advances the concept of space optimization by varying node sizes according to capacity needs. Leis, Kemper, and Neumann prove that, as a result, storage usage for keys of any length is restricted to 52 bytes. Additionally, they demonstrate experimentally that the space required for each key can drop to 8.1 bytes [13].

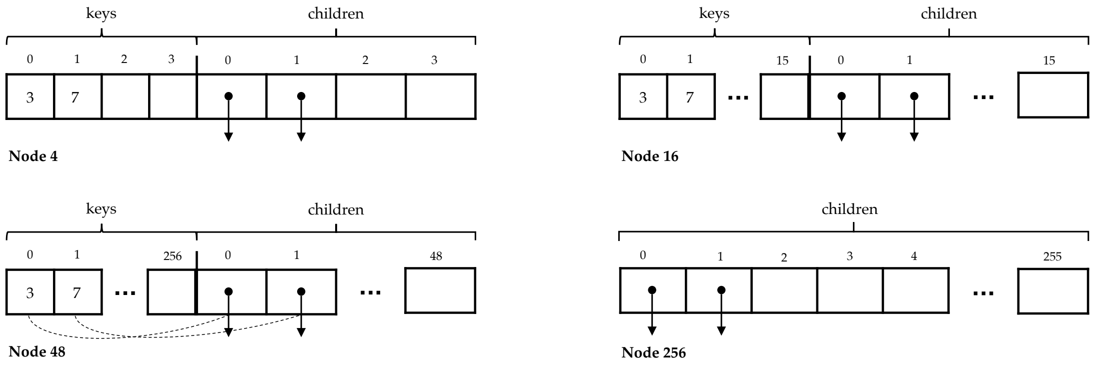

Figure 3.2.: Different nodes available in the ART from [13]

The radix tree consists of outer nodes that store or reference the values and inner nodes that hold pointers to children. A single inner node has a maximum of 2^s children. Here, s represents the span, i.e., the number of bits of the partial key in the node. The span is the same for all nodes, resulting in a height of $\lceil \frac{k}{s} \rceil$. The ART uses a span of 8 bits, leading to a maximum node size of 256. Nevertheless, not all nodes are required to store 256 keys. Therefore, the ART defines four node types based on the demanded capacity. Nodes resize when they reach the capacity of a smaller node or when a new element needs to be inserted and the node is full. This technique avoids the overhead of resizing nodes after each update while reducing storage requirements. The structure of different nodes is depicted in Figure 3.2:

- Node4 contains a key and pointer array of size four, stored sequentially. The keys represent values between 0 and 255. The position of a key in the keys array corresponds directly to its respective position in the collection of child pointers.
- Node16 has a similar structure to Node4, but with a larger capacity. It can hold up to 16 records, and there is a direct correlation between the positions of the entries in the two arrays. Modern CPUs can employ binary searches or SIMD instructions to optimize lookup times.
- Node48 modifies the structure to increase efficiency by using a 256-element array

for the keys and a 48-element array for the child pointers. Instead of searching, the key directly points to a field in the keys array, which defines an offset to the child pointer array. Therefore Node48 can store up to 48 records.

- Node256 eliminates the keys array, and the size of the child pointers array increases to 256. This node is utilized for 49 to 256 entries, and the key directly indexes the array. The space overhead is reduced by eliminating the need to create an extra array for keys, making this node type highly efficient when operating near its capacity limit.

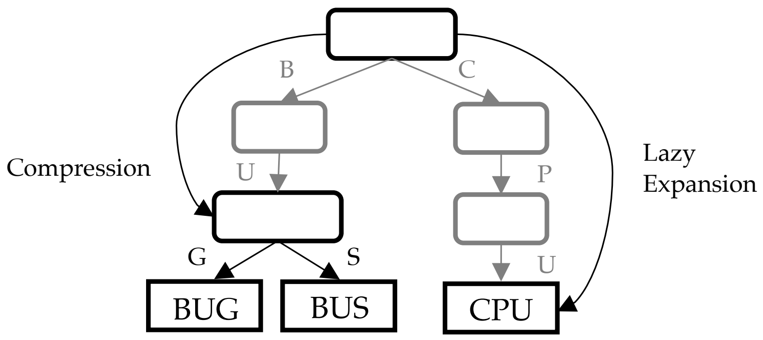

Figure 3.3.: Concept of lazy expansion and path compression

In addition to the adaptive node sizes, the ART incorporates two other concepts which improve memory efficiency: path compression and lazy expansion. To implement these strategies, the header of each node stores additional information, such as the current node size, type, or details about the compressed path [13].

- Lazy expansion states that an inner node is created only when it has a minimum of two children. If not, the node's path is shortened, and all information is stored directly in a leaf. Figure 3.3 depicts the concept for 'CPU'. An internal node forms solely when a key sharing a similar prefix is inserted. Still, the path from the new inner node to the leaf is shortened and stored directly in the outer leaf.

- The path compression technique collapses the path whenever an inner node only has one child, which is illustrated in Figure 3.3. While traversing compressed paths, we have two options to ensure we follow the correct route. In the pessimistic approach, a vector of partial keys skipped at each node is stored and compared to the current key. In contrast, the optimistic approach persists only the number of skipped levels and compares the key at the leaf to validate. ART proposes a hybrid approach where up to 8 bytes of the skipped key remain in the node to compare similarly to the pessimistic case. Any compression beyond this length is treated equally to the optimistic case.

```
def insert(node, key, value):
    if node.leaf:
        node_insert(child, key, value)
    else:
        key = get_partial_key(key, node.depth)
        child = node.get(key)
        if not child:
            lazy_expansion(node, key, value)
        if not child.can_insert():
            child = increase_size(child)
            node_insert(node, child)
        if longest_common_prefix(node, child) != child.depth:
            new_node = Node4()
            node_insert(node, new_node)
            node_insert(new_node, child)
            insert(new_node, key, value)
        else:
            insert(child, key, value)
```

Algorithm 3.1: Inserting into a radix tree

For lookup operations, we compare the search key with the partial keys stored in the node at the current level and then follow the corresponding pointer to the child to reach the correct leaf node.

Algorithm 3.1 presents the pseudocode for inserting into the tree. Similar to the lookup process, we traverse the tree to locate the suitable leaf node for insertion. If the node is a leaf node, we can insert it into the node, provided that we previously checked if the insertion is feasible and that the node is resized when necessary. Otherwise, we obtain the key through the `get_partial_key(node, depth)` function. We use lazy expansion if no path is available from the current key to the node. To create a new leaf node and insert the element, we use the `lazy_expansion(node, key, value)` function. If none of the cases apply, it is necessary to check if insertion into the next node is possible. Otherwise, it is required to increase the size of the node with the `increase_size(child)` method. In the pessimistic approach, to check whether the child node is compressed or not, we need to verify if the shared prefix length between the child node and the key is the same as the node depth using the helper function `longest_common_prefix(node_a, node_b)`. If it is not, a new node must be created in between and inserted into the current node to replace the child. Afterward, we insert the child into the new node and add the element by applying lazy expansion. If none of the above applies, we perform the next recursion step on the child.

Deletion operates similarly to insertion. After deleting an element, specific considerations must be made to maintain the tree's invariants. We resize the leaf node to the next smallest node following deletion if needed. If a leaf node is empty, it has to be removed. As a result, the parent node may require resizing to a smaller node type. Furthermore, it is possible that the parent node now solely contains a single element. To ensure the tree is compressed, we must delete the parent node and reference its child instead in such cases [13].

4. Cache System

This chapter introduces a concept combining features of both disk-based and in-memory DBMS. The concept involves using an in-memory index structure as a cache to an index structure for a disk-based system. This allows increasing amounts of main memory to be used efficiently by the in-memory structure while profiting from less expensive external storage technologies like SSDs.

In the following chapter, we will present the individual components of the system first and then the architecture and information flow. The complete code is available on Github [1].

4.1. System Components

The disk-based system implements storage concepts introduced in Section 2.1.1, with the B+ tree described in Section 2.2.1 as the utilized index structure. The in-memory system is based on the radix tree, as presented in Section 3.2.1. However, the system was simplified, particularly with regard to the disk-based structure. The design is incapable of handling arbitrarily large values. Instead, all keys and stored values are 8-byte integers. In addition, it is not feasible to construct multiple relations, and each key is unique.

4.1.1. Storage Manager

The storage manager is responsible for persisting data outside the main memory, as Section 2.1.1 describes. It saves and retrieves pages on the disk to perform operations in the main memory [9]. The storage manager's implementation uses the host machine's underlying file system. Since only one relation is stored and numerous abstractions are eliminated, pages directly map to locations on the disk. Thus, a single contiguous file is utilized to store all pages sequentially. The relative offset for a page in the file is then calculated by the formula $page\_id * page\_size$. Assigning page_ids in ascending order is essential to minimize space wastage. One may delete a page during execution. This enables the reutilization of both the location in the file and the assigned page_id.

[1]`https://github.com/MatteoWohlrapp/RadixTree`

The storage manager maintains a bitmap to monitor the free space and track any modifications. Storing an intermediate value allows further optimization of the free space search operations by keeping a record of the next available slot. The UML diagram in Figure 4.1 depicts the functionality offered by this component:

`save_page(page)` requires a pointer to a page and copies the entire memory section into the file at the position specified by the page_id in the page header.

`load_page(page, page_id)` loads the data for a given page_id into the specified memory block passed to the function.

`delete_page(page_id)` allows the memory allocated for the page_id to be overwritten.

`get_unused_page_id()` returns the first available page_id.

4.1.2. Buffer Manager

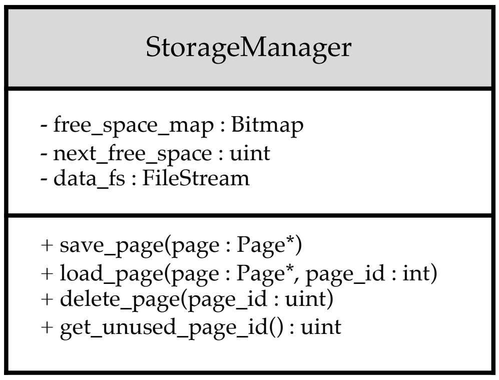
StorageManager
- free_space_map : Bitmap
- next_free_space : uint
- data_fs : FileStream
+ save_page(page : Page*)
+ load_page(page : Page*, page_id : int)
+ delete_page(page_id : uint)
+ get_unused_page_id() : uint

Figure 4.1.: UML diagram of the storage manager

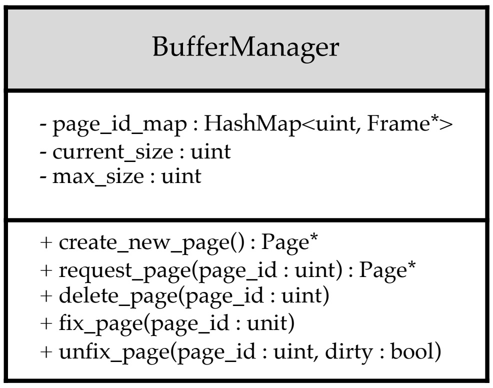
BufferManager
- page_id_map : HashMap<uint, Frame*>
- current_size : uint
- max_size : uint
+ create_new_page() : Page*
+ request_page(page_id : uint) : Page*
+ delete_page(page_id : uint)
+ fix_page(page_id : unit)
+ unfix_page(page_id : uint, dirty : bool)

Figure 4.2.: UML diagram of the buffer manager

The responsibility of the buffer manager is to control the in-memory component of the disk-based system. The tasks of the buffer manager consist of allocating and deallocating new pages in memory and providing information about their location [9]. It offers this functionality by mapping page_ids to frames in a hash map described in Section 2.1.3. The frame is a control structure that includes the page containing the actual information. Additionally, the frame stores the information required to manage the main memory. The memory state is retained using a dirty flag that indicates when the content of the frame's corresponding page has changed. Furthermore, the frame requires a variable to indicate if we currently use the page to prevent premature release of the memory location. Lastly, the frame has a flag for implementing an eviction algorithm. This implementation's buffer capacity is fixed and does not adjust to the system's requirements. To fulfill its responsibilities, the buffer manager provides several functionalities illustrated in Figure 4.2.

`create_new_page()` allocates a new page and writes an unassigned page_id to a dedicated header at the top of the page.

`request_page(page_id)` returns the pointer to the page identified by page_id. If the required page is not in the main memory, we fetch it from the disk.

`delete_page(page_id)` deletes a page from the main memory and the disk. However, we don't deallocate the memory and reuse the address for new pages.

`fix_page(page_id)` locks a page and prevents it from being swapped out of memory. This function is used before a page is accessed and called implicitly when creating or requesting a page.

`unfix_page(page_id, dirty)` method unlocks a page and signals whether it has been modified through the dirty flag parameter.

When the buffer is full, and we request a page that is not currently available in the main memory, or when a new page is created, and the buffer is full, it becomes necessary to load the page from the disk, replacing an existing page in the buffer. The replacement strategy is crucial in determining the system's performance due to the access gap between the main memory and external storage. Recently accessed pages are more likely to be used again soon. This observation implies that an approach based on the expected reuse of pages is more effective than random access. One frequently used strategy is the Least Recently Used (LRU) algorithm, which removes the page that was accessed least recently [9]. This implementation uses a simple heuristic:

The initial step is to choose a page from the buffer randomly. Afterward, we verify whether the page is unfixed and has had recent access. The marked flag determines this. If the page was recently accessed, we remove the mark, and the search is conducted again. If not, we remove the page_id from the map, and if the page has been altered, save it to memory. Finally, we return the pointer to the frame for further use to avoid reallocation.

4.1.3. B+ Tree

As previously noted in Section 2.2.1, the B+ tree indexes keys and their corresponding pages, which the buffer and storage manager control. The tree comprises inner nodes holding the keys and page_ids of their children and outer nodes, or leaves containing all keys and their corresponding values. Limiting the information to only the outer nodes causes an increase in fan-out and a decrease in the overall depth of the tree [9]. In this implementation, each node represents a page in the buffer manager.

The nodes are implemented as structs. When pages are swapped in and out of external storage, they must be reconstructed from a memory block when loaded from a disk to facilitate operations. A flag in the page header specifies the node type and is used for identifying and casting the pointer to the appropriate data structure. The 'PAGE_SIZE' template variable determines the size of a page and, subsequently, the node's size during compile time. The number of keys and children in an inner node, or keys and values in an outer node, is adjusted following the page's size.

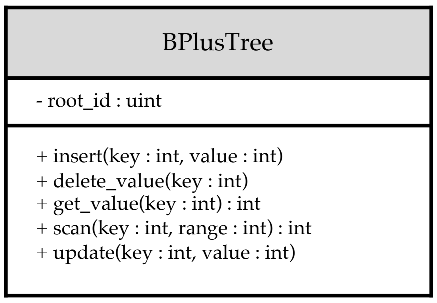
BPlusTree
- root_id : uint
+ insert(key : int, value : int)
+ delete_value(key : int)
+ get_value(key : int) : int
+ scan(key : int, range : int) : int
+ update(key : int, value : int)

Figure 4.3.: UML diagram of the B+ tree

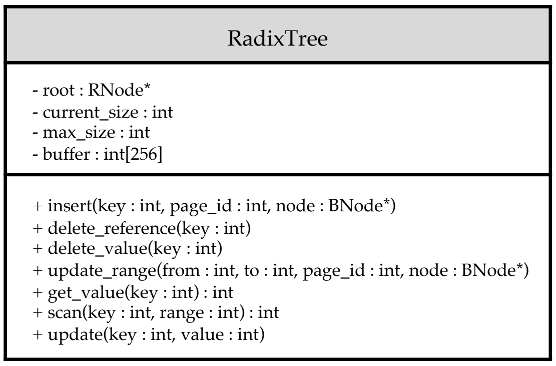
RadixTree
- root : RNode*
- current_size : int
- max_size : int
- buffer : int[256]
+ insert(key : int, page_id : int, node : BNode*)
+ delete_reference(key : int)
+ delete_value(key : int)
+ update_range(from : int, to : int, page_id : int, node : BNode*)
+ get_value(key : int) : int
+ scan(key : int, range : int) : int
+ update(key : int, value : int)

Figure 4.4.: UML diagram of the radix tree

Figure 4.3 illustrates the interface for the B+ tree, which offers operations for standard database functionality. Moreover, it stores the page_id of the root node. Direct storage of a pointer to the root is not feasible due to possible changes in memory location caused by disk pages replacing main memory pages. When navigating the tree, requesting the next page_id from the buffer manager is required.

`insert(key, value)` adds a key-value pair into the tree. This procedure follows the description in Section 2.2.1. The system inherits the existing scheme, but the implementation includes a slight modification. The node-splitting process differs from the conventional bottom-up method, as we split nodes top-down precautiously when full. This approach may result in splitting, even when it is unnecessary. However, it enables page locking from the root downwards.

`delete_value(key)` removes a key-value pair from the tree. The method is similar to that described in Section 2.2.1, which involves substitution or merging if the node does not have enough elements to delete. Nonetheless, like the insert operation, the underpopulated nodes are already addressed as we traverse the tree, allowing nodes to be locked from the top down.

`get_value(key)` returns the value corresponding to the key. If the value does not exist, we return the minimum value for an 8-byte integer to indicate its unavailability.

`scan(key, range)` starts at the value corresponding to the key and then performs XOR operations on the subsequent 'range' amounts of values. If the returned value equals $INT64\_MIN$, the function increments it by one, as the maximum negative value would indicate that the key is unavailable.

`update(key, value)` updates the value field of a key if present.

4.1.4. Radix Tree

The cache structure utilized here is a radix tree, based on the concept presented in Section 3.2.1. However, we made some modifications to adapt it to the new purpose. As the tree serves cache functionality, it does not save the value corresponding to the key. Instead, it stores the page_id and a pointer to the B+ node containing it. This is because the page might have been replaced since our last access. To ensure page correctness, we compare the page_id to the id in the page's header. When we delete a page, the areas in the main memory are not freed, which means the page references remain valid, with no potential for malicious memory access. As the system uses short keys with a length of only 8 bytes and employs a pessimistic comparison approach, each node stores the complete key. We transform the key to prevent sign-bit problems by adding $INT64\_MAX + 1$ and casting it to an unsigned integer. This ensures the preservation of order. Subsequently, the inverse function must be applied to retrieve the B+ tree values using the key. Figure 4.4 displays the available operations for the tree. In the illustration, BNode refers to a node in the B+ tree, while RNode specifies a radix tree node:

`insert(key, page_id, page)` adds a reference to the node of the key along with the corresponding page_id. This operation follows a process close to the pseudocode described in Section 3.2.1. If there is no path from an inner node to a root, we perform lazy expansion while we combine nodes with similar prefixes by compressing them.

`delete_reference(key)` removes the reference for the key from the tree. Deletion follows a process similar to the deletion process for adaptive radix trees, described in Section 3.2.1. Any empty nodes are removed after deleting an element from the tree. If applicable, we clear references to compress the tree again.

`delete_value(key)` removes the key from its position in the B+ tree. Afterward, we remove the reference to the node from the cache by calling `delete_reference(key)`.

`update_range(from, to, page_id, node)` updates the page_id and node reference for all keys k, $from \leq k \leq to$. The ordered nature of the keys makes the process efficient by traversing only the subtree covering the range of values. Figure 4.5 provides an example of a subtree, highlighted in blue, that undergoes the update when performing the operation for all values between 'BUG' and 'BUS'.

`get_value(key)` retrieves the value stored in the B+ tree corresponding to the provided key. If the element is absent in the cache or the node, the function returns $INT64\_MIN$.

`scan(key, range)` attempts to locate the key's position in the B+ tree through its reference in the cache. If found, we execute the same operation performed in the B+ tree. 'Range' XOR operations are applied to the values beginning at the position of the key.

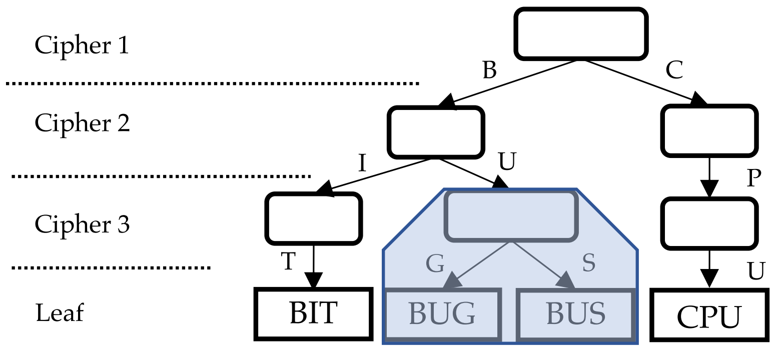

Figure 4.5.: Range update on a radix tree

To manage the cache size, we maintain a variable that tracks the number of bytes used. We adjust it when operations modify the tree size, such as insertion, resizing, or deletion. If the cache reaches its upper size limit and a new item needs to be inserted, we remove a previously inserted item to make room for future insertions. A ring buffer that stores up to 256 of the most recently inserted elements has been implemented to achieve this objective. During deletion, we remove the first inserted value. However, this method provides only a basic heuristic, as there is no guarantee that the items will still be cached, and the buffer size is relatively small compared to the number of items in the tree.

4.2. Architecture

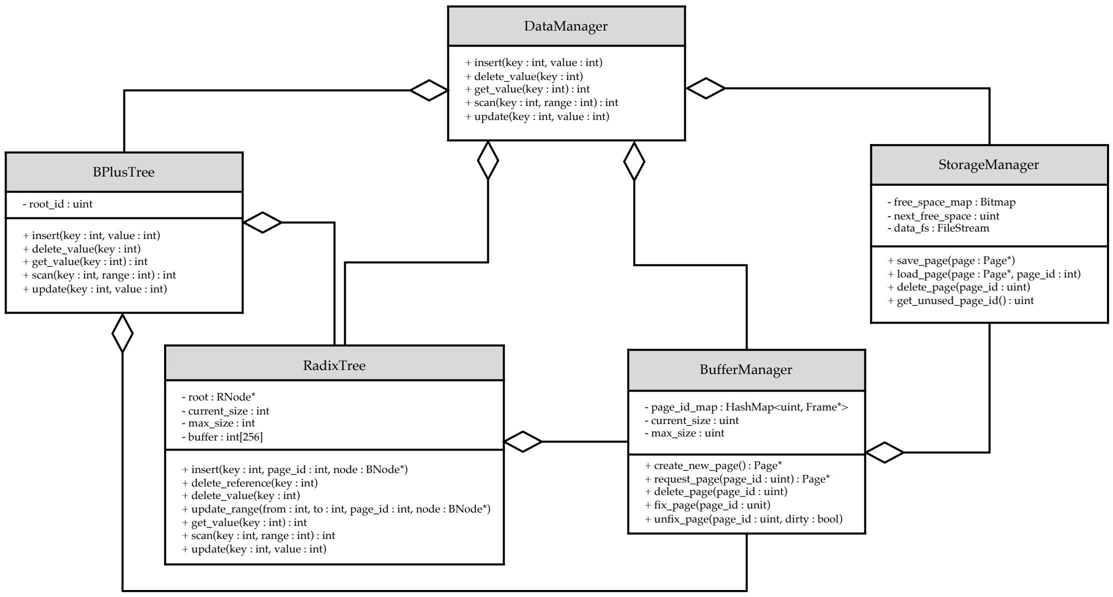
DataManager
+ insert(key : int, value : int)
+ delete_value(key : int)
+ get_value(key : int) : int
+ scan(key : int, range : int) : int
+ update(key : int, value : int)
BPlusTree
- root_id : uint
+ insert(key : int, value : int)
+ delete_value(key : int)
+ get_value(key : int) : int
+ scan(key : int, range : int) : int
+ update(key : int, value : int)
StorageManager
- free_space_map : Bitmap
- next_free_space : uint
- data_fs : FileStream
+ save_page(page : Page*)
+ load_page(page : Page*, page_id : int)
+ delete_page(page_id : uint)
+ get_unused_page_id() : uint
RadixTree
- root : RNode*
- current_size : int
- max_size : int
- buffer : int[256]
+ insert(key : int, page_id : int, node : BNode*)
+ delete_reference(key : int)
+ delete_value(key : int)
+ update_range(from : int, to : int, page_id : int, node : BNode*)
+ get_value(key : int) : int
+ scan(key : int, range : int) : int
+ update(key : int, value : int)
BufferManager
- page_id_map : HashMap<uint, Frame*>
- current_size : uint
- max_size : uint
+ create_new_page() : Page*
+ request_page(page_id : uint) : Page*
+ delete_page(page_id : uint)
+ fix_page(page_id : unit)
+ unfix_page(page_id : uint, dirty : bool)

Figure 4.6.: UML diagram of the database system

To integrate all the components presented above, we introduce a new structure. Acting as the central hub, the `DataManager` creates the various composites at startup and manages memory deallocation at shutdown. These components include the `BufferManager`, `StorageManager`, `BPlusTree` and `RadixTree`. To exchange pages between memory and disk, the buffer manager holds a reference to the storage manager. Both the radix tree and the B+ tree maintain a reference to the buffer manager. The radix tree utilizes the reference to fix pages while performing operations on the referrenced nodes of the B+ tree. The B+ tree additionally uses the reference to create and request pages. The B+ tree aggregates the radix tree to add particular page information to the cache. In general, the data manager provides an interface to the functionality of the database system and manages the interaction between the cache and the B+ tree, which we will discuss in more detail in the following section.

4.2.1. Insert

```
def insert(key, value):
    bplus_tree.insert(key, value)
```

Algorithm 4.1: Insert

The system does not use the cache to enhance performance during insertion. The data manager merely inserts the key-value pair into the B+ tree, as presented in Algorithm 4.1.

Nevertheless, when we insert a key-value pair, it gets stored in the cache. Upon inserting into an outer leaf of the B+ tree, the key, page_id, and page pointer are all input into the cache.

During B+ tree insertion, we may need to split the outer node because it is full. This procedure is explained in Section 2.2.1. Consequently, half of the key-value pairs within the node will be relocated to another node. Thus, all of these values that were previously cached become invalid. However, we do not remove the previous page, so the entries would remain in the cache without contributing any correct lookups. Therefore, we use the `update_range(from, to, page_id, node)` function introduced earlier to update all transferred elements.

4.2.2. Read

```
def get_value(key):
    if radix_tree != NULL:
        value = radix_tree.get_value(key)
        if value != INT64_MIN:
            return value
    return bplus_tree.get_value(key)
```

Algorithm 4.2: Read

The database system reads data from the cache according to Algorithm 4.2 for the initial lookup. If the returned value does not equal *INT64_MIN*, the operation succeeds, and we return the item.

Otherwise, it indicates we cannot find the element in the cache. An entry may exist for the corresponding key, but the stored page_id does not match the page_id of the referenced page. Such a situation occurs when the buffer manager has swapped out the page. Removing the entry from the tree, in this case, will prevent further searches in the wrong location. Alternatively, the return of *INT64_MIN* could indicate that the reference has yet to be inserted into the tree. In either case, the B+ tree's read function is invoked. After reaching an outer node, we add the key and the corresponding page to the cache before returning the value.

4.2.3. Update

The pseudocode for the update operation is shown in Algorithm 4.3. We first perform an update operation on the cache. The radix tree tries to locate the page. If successful, it converts the page to an outer node and completes the update operation using a function specified by the node of the B+ tree. If successful, this function call returns true, indicating that no further actions are required.

Otherwise, we need to carry out the update operation on the B+ tree, which adds the relevant elements to the cache before updating the value in the tree.

```
def update(key, value):
    if radix_tree != NULL:
        if radix_tree.update(key, value) == true:
            return
    bplus_tree.update(key, value)
```

Algorithm 4.3: Update

4.2.4. Scan

The cache is used to find the starting element for the scanning process. If the element is referenced within the limits of the radix tree, we carry out a scan operation equal to that of a B+ tree to execute 'range' XOR operations on the tree. If the call fails to retrieve any value, *INT64_MIN* is returned.

In that case, we scan the B+ tree and insert the starting position into the cache.

```
def scan(key, range):
    if radix_tree != NULL:
        value = radix_tree.scan(key, range)
        if value != INT64_MIN:
            return value
    return bplus_tree.scan(key, range)
```

Algorithm 4.4: Scan

4.2.5. Delete

```
def delete_value(key):
    if radix_tree != NULL:
        if radix_tree.delete_value(key) == true:
            return
    bplus_tree.delete_value(key)
```

Algorithm 4.5: Delete

When deleting, we first attempt to remove the item through the cache according to Algorithm 4.5. If successful, we return true, and no further actions are required. In addition to the regular restrictions of matching page_id with the referenced page_id in the leaf of the page, it is necessary to evaluate the feasibility of removing the key-value pair from the B+ node. When there are insufficient elements in the node, as detailed in Section 2.2.1, substitution or merging is required, which is out of the scope of the cache.

If necessary, we then operate on the B+ tree, which deletes the reference from the cache upon removal. Substitution involves shifting only one element from node to node and may be resolved with a simple insertion or update in the cache. However, merging key-value pairs requires transferring multiple elements. To ensure the validity of keys in the cache, we use `update_range(from, to, page_id, node)` in these cases.

5. Evaluation

This chapter provides benchmarks for the introduced database system. First, we evaluate the performance when increasing the number of entries in the system. Then, we change the distribution and finally analyze different memory allocation proportions between the cache and the buffer.

5.1. Setup

The benchmarks were conducted on an x86_64 system architecture. The system features 20 Intel Core i9-7900X CPUs, clocking up to 3.30GHz. Each CPU has one socket with ten cores, each operating two threads. The cumulative cache includes an L1 data cache of 320 KiB, an L1 instruction cache of 320 KiB, an L2 cache of 10 MiB, and an L3 cache of 13.8 MiB. Overall, the machine has 125 GB of main memory. The executable program was compiled with GCC 12.2.0 and C++17 on Ubuntu 22.10.

5.2. Experiments

In order to evaluate the system's performance, we conducted benchmarks based on the Yahoo! Cloud Serving Benchmark (YCSB). The YCSB benchmark was created to establish performance comparability between different database systems [4]. It includes several standardized workloads with fixed proportions for various operations that mirror real-world use cases. We replicated workloads A, B, C, and E from the YCSB repository [1]. Furthermore, we created workload X to evaluate the delete performance of the system. Table 5.1 presents the proportion of operations for the different workloads.

For each measurement, a vector of uniformly distributed values from `INT64_MIN + 1` to `INT64_MAX`, using the `std::uniform_int_distribution` [2] and a `std::random_device` [3] is created and inserted to populate the database at the beginning. Then a `std::discrete_distribution` [4] selects the operations and stores them in a

[1]`https://github.com/brianfrankcooper/YCSB`

[2]`https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution`

[3]`https://en.cppreference.com/w/cpp/numeric/random/random_device`

[4]`https://en.cppreference.com/w/cpp/numeric/random/discrete_distribution`

Workload	A	B	C	E	X
Insert proportion	0	0	0	0	0
Read proportion	0.5	0.95	1	0.95	0.9
Update proportion	0.5	0.05	0	0	0
Scan proportion	0	0	0	0.05	0
Delete proportion	0	0	0	0	0.1

Table 5.1.: Operation proportions for the YCSB workloads

separate vector. The last step is to select the indices that refer to the values used for the operations. Depending on the configuration, either `std::uniform_int_distribution` [2] or `std::geometric_distribution` [5] is employed. We store the indices again in a vector. Note that the distribution for the indices does not correlate with the range of values requested, as they have been chosen randomly and are not sorted. While the benchmark runs, we iterate over the operations to reduce generation overhead during execution.

Using the `std::chrono::high_resolution_clock` [6], all operations are timed individually. This approach allows us to obtain the overall throughput and additionally provides more detailed information, such as the mean, median, and percentiles for each type of operation.

To verify the results, we ran each benchmark and later examined the status of the radix- and B+ trees to validate their respective invariants. We also checked that the correct elements were updated or deleted and that all other elements were still in the database. Furthermore, Valgrind's 'callgrind' tool was used to ensure the cache had the expected behavior, with the tool showing a reduction in operations called on the B+ tree when the cache was activated.

5.2.1. Varying Record Count

This experiment changes the number of entries or records in the system. For each workload, we present two graphs with detailed information. One chart shows the throughput in operations per second on the y-axis, with the number of records varying from 2 to 10 million in increments of 2 million on the x-axis. The second graph visualizes the mean, median, 90th, 95th, and 99th percentiles of execution time per operation type of the workload for the run with the highest number of records. Both graphs evaluate performance with the cache component enabled and disabled. While

[5]`https://en.cppreference.com/w/cpp/numeric/random/geometric_distribution`
[6]`https://en.cppreference.com/w/cpp/chrono/high_resolution_clock`

the record size is varied, the rest of the parameters remained constant throughout the test, as shown in table 5.2. The benchmark is run on a single thread. Appendix A.0.1 shows a chart with multi-threaded performance. However, we could not optimize the locking mechanism to improve the performance beyond that of a single thread.

Operation Count	Buffer Size [B]	Cache Size [B]	Distribution	Coefficient
20,000,000	163,840,000	697,932,185	Geometric	0.001

Table 5.2.: Parameters for the benchmark varying the record count

The buffer reaches its main memory capacity after inserting ≈ 5 million elements, while the radix tree fills up after inserting ≈ 6.6 million records.

For this experiment, we expect an overall decrease in performance as the number of entries increases. In addition, we expect the read and update operations to perform better for the cached version, as they benefit most from the implementation. For scan, the proportion of activity involving the cache is lower, while for delete, there is additional overhead in managing the radix tree when deleting items. Insert is expected to perform worst as there is no performance benefit from the cache, just additional cost in maintaining the tree.

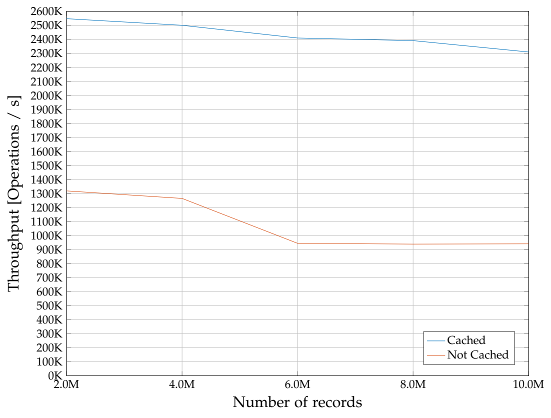

Figure 5.1.: Workload A throughput

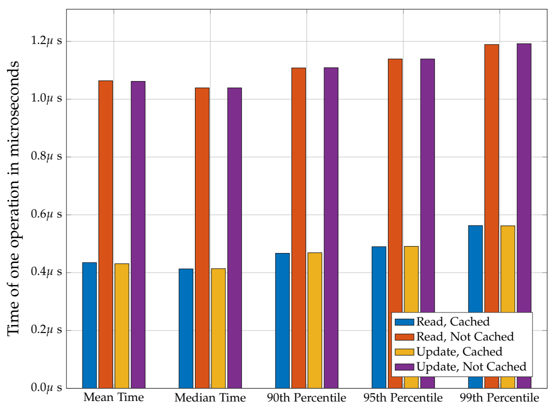

Figure 5.2.: Workload A timings

Figure 5.1 shows that the cached configuration outperforms the non-cached system for all record counts tested in workload A. The throughput starts at 2.55 million operations per second for 2 million records. As the number of records increases, the throughput drops slightly to 2.31 million operations per second for 10 million records.

In comparison, the uncached configuration shows a stronger drop in throughput as the number of records increases. It starts with a throughput of just over 1.32 million for 2 million records and drops slightly to around 941 thousand operations per second for 10 million records. The most significant reduction occurs between 4 and 6 million keys inserted. Overall, the cached system performs about twice as well as the system with the cache disabled.

Looking at the individual operations in Figure 5.2, we can see that cached operations have a lower average execution time than non-cached operations. Both read and update operations perform similarly across all measured variables, including mean, median, 90th, 95th, and 99th percentiles. For the cached version, the operation times are approximately half those of the non-cached system. Enabling the cache results in operation times between 0.43 and 0.44 µs, while not using it results in a mean operation time of 1.06 µs. Generally, the mean time of operations is relatively close to the 99th percentile.

The decrease in throughput as the number of elements increases in Figure 5.1 is expected. The I/O operations associated with the full buffer can lead to performance degradation for both implementations. Notably, the performance without cache drops most prominently when the first I/O operations are required, which happens at around 5 million entries. After this decrease, however, the throughput remains relatively stable, which could be related to the locality of the pages in the buffer. We used a geometric distribution for this benchmark, skewing the probability for the accessed indices, meaning we need to access fewer nodes in the B+ tree during the operations. In the cached solution, this is not as noticeable, which could be related to the fact that the cache avoids the need to request the internal nodes of the B+ tree, resulting in a smaller number of swapped pages. The fact that read or update operations fix and mark the outer nodes of the B+ tree underlines that, which further reduces the likelihood of these nodes being subject to swapping operations, in line with the replacement strategy described in Section 4.1.2.

Given the implementation, it is unexpected that the 99th percentile for both read and update operations in 5.2 is slower in the uncached implementation. Cache misses are costly because they require processing on the radix- and B+ trees. A possible explanation could be that the potential reduction in disk swaps leads to increased page locality in the buffer manager. Furthermore, although the mean and 99th percentile are very similar, it could still be that a few outlying operations took longer for the cached system than for the uncached system but that the total number of cache misses was small.

The throughput of workload B, shown in Figure 5.3, is comparable to that of workload

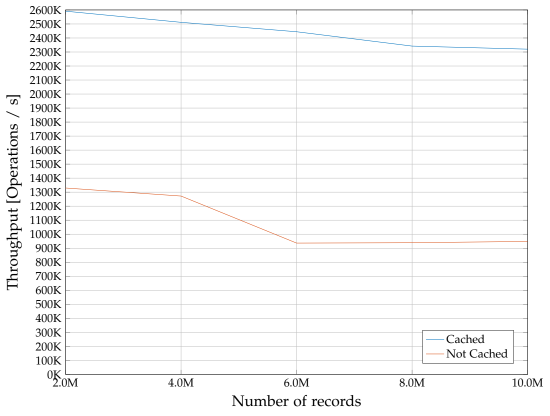

Figure 5.3.: Workload B throughput

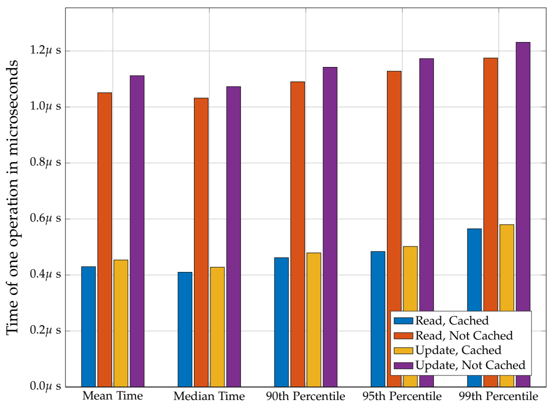

Figure 5.4.: Workload B timings

A. For the cached system, the throughput starts at 2.59 million operations per second for 2 million entries and drops to 2.32 million for 10 million records. In contrast, the solution without cache is about half as fast, starting at 1.33 million operations per second and then dropping to a throughput of 949 thousand. Again, the most significant decrease is around the number of records where the buffer reaches its main memory capacity.

Looking at Figure 5.4, cached operations outperform non-cached operations across all measurements. For updates and reads, the average time of the operations is between 0.43 and 0.45 µs in the cached implementation. The non-cached implementation is slower, with a mean of 1.08 µs over both operations. As before, the mean and the 99th percentile difference is insignificant.

The update and read operations perform relatively similarly for workload A. The only change between the workloads is the proportion of update and read. The result was expected following the explanation for workload A.

Workload C consists only of read operations, and its throughput, shown in Figure 5.5, is very similar to workloads A and B. For the cached solution, a throughput of 2.61 million operations per second is achieved for 2 million entries. This number drops to 2.36 million operations per second for 10 million keys inserted. In contrast, the non-cached solution starts at 1.34 million operations and drops to 917 thousand for 10 million records, with the sharpest reduction between 4 and 6 million records.

The individual operation times are also very similar, as shown in Figure 5.6, with an average operation time of 0.42 µs for the cached implementation. In comparison, the non-cached operations take 1.09 µs on average. Overall, the difference between the

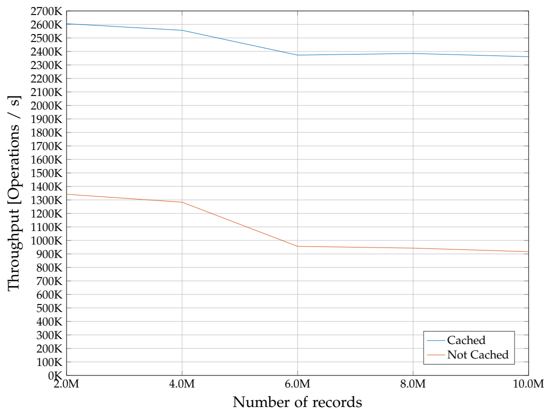

Figure 5.5.: Workload C throughput

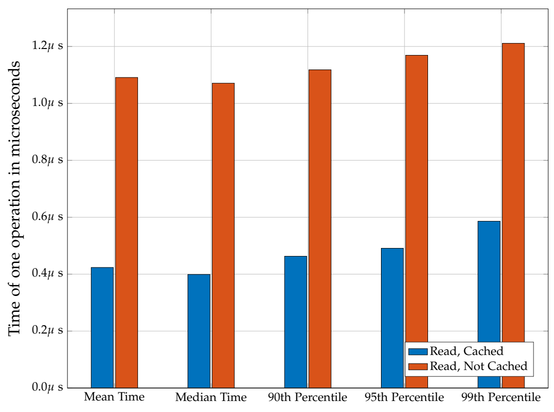

Figure 5.6.: Workload C timings

mean and the 99th percentile is not noteworthy.

Again, following the explanation of workload A, the decrease in throughput as the number of entries increases is expected. In contrast, we did not anticipate the significant difference between cached and non-cached execution times for the 99th percentile. However, it can be explained by a few expensive outliers and generally few cache misses.

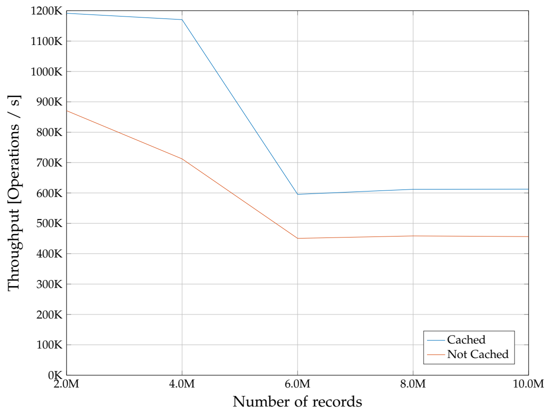

Figure 5.7.: Workload E throughput

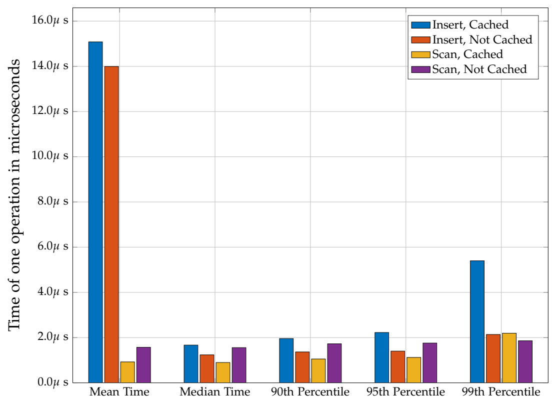

Figure 5.8.: Workload E timings

Figure 5.7 shows an overall lower throughput for workload E compared to the previous workloads. For the cached solution, 2 million records correspond to a throughput of 1.19 million operations per second, less than half that of workloads A-C. This number remains relatively constant for the first record increase before dropping significantly at 6 million operations. It then remains stable at a throughput of 612 thousand until 10 million records are reached. The non-cached system shows a similar trend, with 871 thousand operations per second initially, dropping to 451 thousand for 6 million records, remaining almost constant afterward.

The statistics for individual operations in Figure 5.8 show that while the scan operation times are in a similar magnitude to the read and update operations, with an average of 0.93 µs per operation and no significant increases for the higher percentiles, the insert operation performs significantly worse. Insertion is the first example of a cached function slower than the non-cached one. With cache enabled, the average insert operation takes 15.08 µs, while with cache disabled, it takes 13.99 µs. While these values are relatively similar, the 99th percentile is particularly noteworthy. For the cached system, it is 5.4 µs, while for the non-cached system, it is 2.19 µs, showing a big difference. Also, the 99th percentile is faster than the mean for both cached and non-cached, meaning there are few but very slow outliers.

For this workload, the difference between cached and non-cached is smaller than in the previous workloads. This is because although only 5% of the operations are insert operations, they are significantly slower than the scan operations, which minimizes their speedup effect. The fact that insert operations are slower is no surprise, as they do not use the cache to accelerate operations at all but rather increase the overhead by building the radix tree simultaneously. The steep drop in throughput between 4 and 6 million entries is probably due to the buffer reaching its capacity. In particular, insert operations, which in the worst case need to access more nodes and therefore require more I/O, can be costly. The fact that the mean time for operations is slower than the 99th percentile could be related to the cascading effect of inserts. Very rarely, splitting most of the nodes as we move down the tree may be necessary, significantly increasing the operation time compared to insertion at the leaf.

In workload X, the overall performance of the cached implementation is slower than the non-cached implementation for the first time, as shown in Figure 5.9. The cached solution has an initial throughput of 1.38 million operations per second for 2 million records, then drops to 976 thousand operations per second above 6 million records, remaining relatively stable afterward. On the other hand, the version with the cache disabled starts with a throughput of 1.43 million before dropping to 1.01 million operations per second between 4 and 6 million records to stabilize.

Looking at the individual operations in Figure 5.10, both read and delete perform worse for the cached solution. Delete operations are significantly faster than insert operations for cached and non-cached cases. While the mean insert time is 15.08 µs, as shown in Figure 5.10, the mean delete time is 1.20 µs for the cached and 1.11 µs for the non-cached implementation. An interesting observation is that the mean read time for the cached implementation is higher at 1.03 µs compared to 1 microsecond in the non-cached one.

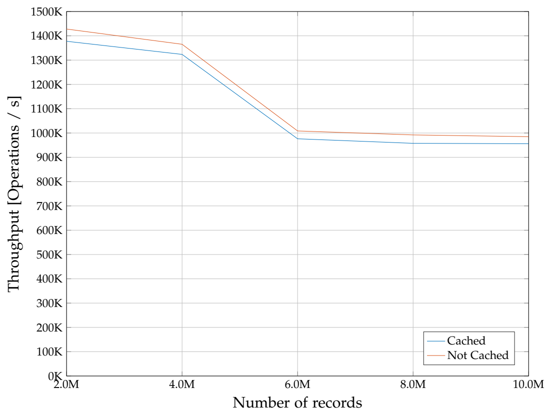

Figure 5.9.: Workload X throughput

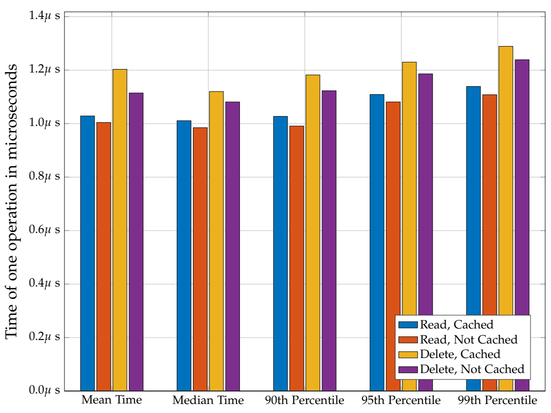

Figure 5.10.: Workload X timings

The higher mean operation time for the read operation is particularly surprising. Without deleting, the read operation performs much better. It could be connected to the fact that the cache is useless whenever we read previously deleted items, which is potentially a lot in this workload. This is because the lookup is first performed in the radix tree, returning $INT64\_MIN$ if the item is not in the cache. This signals that the operation will be performed again in the B+ tree, and additionally, we need to delete the reference in the radix tree. The general problem with caches is that they allow for false positives, where the element does not exist, but the system cannot differentiate. The discrepancy between cached and non-cached for deletes is less significant than for inserts. While in some delete cases, it may cause overhead to manage the cache, in other cases, the cache may be helpful to delete items from the system without accessing the B+ tree, closing the gap between the two implementations.

An important note about the graphs in this section is that while the radix tree size was limited, most of the indexed values were still in the cache. This is because we insert the values into the B+ and radix trees before we execute the workloads. Since we use a geometric distribution, which skews the indexes being accessed towards the first values that were inserted, most of the requested values were likely inserted into the cache before it was full. Nevertheless, this effect should be achievable with a more sophisticated approach to cache eviction and was deliberately used here to support the simple strategy applied in the implementation.

5.2.2. Varying Index Distribution

The first part of this experiment varies the distribution used to select the indices. Figures 5.11 and 5.12 switch between uniform and geometric distributions for both cached and non-cached configurations. The y-axis of each figure shows the throughput in operations per second, while the x-axis groups the different distributions for the various workloads. The rest of the parameters remain constant throughout the test, as shown in Table 5.3. For this experiment, the number of operations and records is reduced compared to the first experiment because the uniform distribution takes much longer to run. In addition, values are inserted in reverse order, causing the cache to fill with values unlikely to be accessed by the geometric distribution initially. This means the cache heuristic is vital to store the most recently accessed items during execution. The experiment runs in a single thread.

Record Count	Operation Count	Buffer Size [B]	Cache Size [B]	Coefficient
1,000,000	1,000,000	16,384,000	69,793,215	0.1

Table 5.3.: Parameters for the benchmark varying the distribution

The buffer reaches its main memory capacity after inserting ≈ 500 thousand elements, while the radix tree fills up after inserting ≈ 660 thousand records.

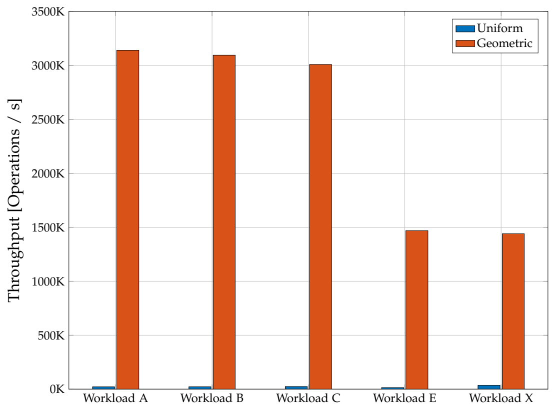

Figure 5.11.: Throughput for different distributions and workloads, cached

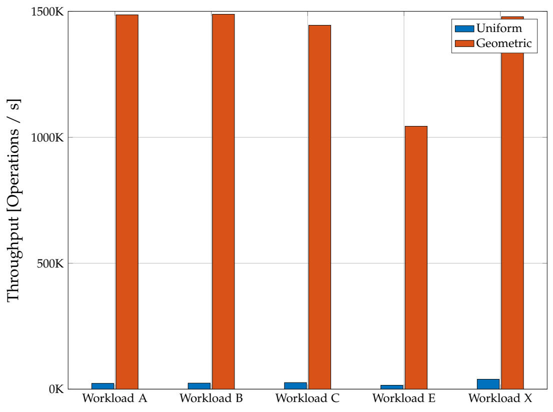

Figure 5.12.: Throughput for different distributions and workloads, not cached

For this experiment, we expect the uniform distribution to perform worse than the geometric distribution. Uniform access makes caching less effective, which should also be reflected in the difference in throughput between the cached and non-cached systems.

Analyzing Figure 5.11, the speed of the geometric distribution is comparable to the results described in Section 5.2.1. For the cached version, workload A-C has a relatively high throughput, reaching over 3.01 million operations per second. In comparison, workload A-C in the non-cached configuration is consistent with the previous data, achieving about half the throughput of the cached system. For workload E, the cached version is still faster at 1.47 million operations per second, but the difference is reduced, with the non-cached version having a throughput of 1.04 million. Workload X again performs better uncached. In contrast, the uniform distribution behaves much worse. The throughput of the cached version ranges from 15 to 36 thousand operations per second. For the uncached system, the throughput is only slightly higher, ranging from 16 to 39 thousand.

Although the buffer was already full when the items likely to be requested in the benchmark were inserted, the cache eviction strategy seemed sufficient for a coefficient of 0.1 in the geometric distribution. If the requests are skewed, it is more likely that the correct pages are already in memory. Therefore, there is no need to load them from the disk. However, with uniform access, values may be requested from pages that still need to be added to the main memory and, therefore, more expensive to look up. For the cache, this means that even when the values are in the radix tree, the page_id will most likely not match the referenced page, and a lookup in the B+ tree must be performed. While this behavior was expected, the difference is significant and underlines how disk access is a major contributor to increased runtime. It is also worth noting that the number of elements is relatively small, resulting in a shallow B+ tree. The effect of I/O operations would be even more significant with a deeper tree, as we would need to access and load more pages.

In addition to changing the distributions, the second part of the experiment varies the probability coefficient of the geometric distribution. Four different coefficients are tested, ranging from 0.0009 to 0.9. The results are shown in Figures 5.14 and 5.15. The y-axis of each figure shows the throughput in thousand operations per second, while the x-axis groups the different workloads by the coefficients. The rest of the parameters remained constant throughout the test, as shown in Table 5.4. For this experiment, the number of operations and data sets is also reduced. In addition, we insert the values in reverse order and run the benchmarks in a single thread.

Record Count	Operation Count	Buffer Size [B]	Cache Size [B]	Distribution
1,000,000	1,000,000	16,384,000	69,793,215	Geometric

Table 5.4.: Parameters for the benchmark varying the coefficient for the geometric distribution

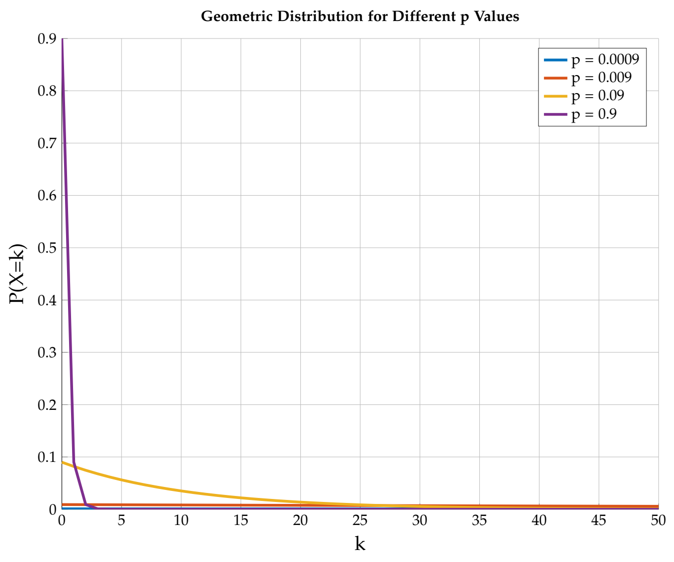

Figure 5.13.: Geometric distribution for different coefficients

The coefficients, or probability p, for the geometric distribution can take values between 0 and 1, and the effect on the distribution can be observed in Figure 5.13. The closer the probability is to 1, the more skewed the distribution is.

For this experiment, we expect to see an increase in performance with a higher index locality. Because the implemented cache eviction strategy has limitations, more distributed indices should reduce throughput.

For the cached implementation, shown in Figure 5.14, the probability in the distribution has a significant effect. For $p = 0.9$ the throughput is almost four times higher than for $p = 0.0009$. $P = 0.0009$ results in throughput for workloads A-C of below 815 thousand, while workload E performs slightly worse, and workload X outperforms the others. Increasing the coefficient by factor 10, the read/update heavy workloads have a higher throughput of around 1.5 million operations per second. In contrast, workload E performs worst with a throughput of 789 thousand, while workload X

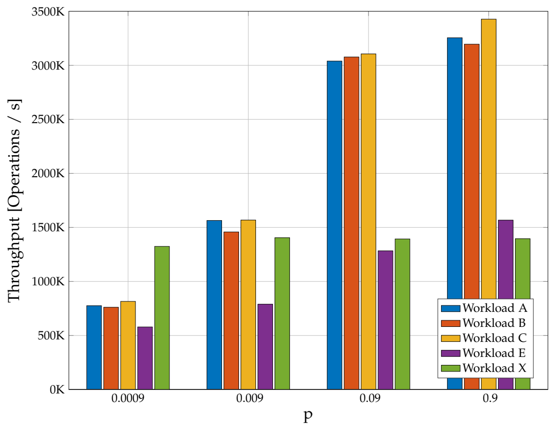

Figure 5.14.: Throughput for different coefficients p of the geometric distribution, cached

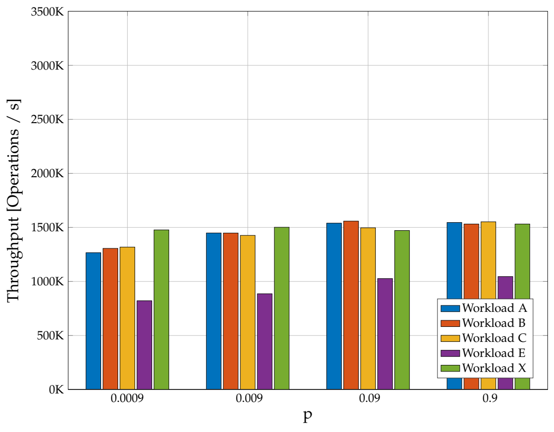

Figure 5.15.: Throughput for different coefficients p of the geometric distribution, not cached

remains relatively stable. The throughput remains stable for $p = 0.09$ and $p = 0.009$, showing similar figures as the previous benchmarks with a throughput of over 3 million for workloads A-C. Workload E performs slightly better than before, with throughputs between 1.28 and 1.57 million, while workload X remains stable.

The geometric distribution coefficient has less impact on the non-cached implementation, as shown in Figure 5.15. For all the values tested, workloads A-C and X remain between 1.27 and 1.55 million operations per second, while workload E performs slightly worse with a throughput of 822 to 1.05 million.

Overall, this is what is expected from this experiment. The buffer manager has a more sophisticated eviction algorithm than the radix tree, using an LRU heuristic. This means it may perform better on values that are not as skewed because it can keep more of the relevant information in the main memory. In contrast, the ring buffer used in the radix tree only has a size of 256. When the buffer is full, we attempt to delete the last of these elements. However, because the ring buffer is small compared to the size of the radix tree, this value may be requested again in the skewed distribution but is no longer available in the cache. As a result, the cache is not as effective. The fact that although the smallest coefficient is very similar to that used for the benchmarks in Figures 5.1 to 5.10, but the read and update focused workloads A-C perform much worse underlines that. This is because the elements are inserted in reverse order for this benchmark, filling the cache before the most frequently requested values are added. This may not be a problem for higher coefficients, as the 256-byte buffer is sufficient to keep all the hot keys in the tree, but for smaller p, it does not seem to be a good enough approximation. It is also surprising that workload X is relatively stable in the cached implementation, outperforming workloads A-C for the smallest coefficient. This could be related to the inefficiency of the cache when reading deleted values, causing more execution in the B+ tree which is less affected, as described above. It is also interesting to note that workload E has increasing performance with higher probability p. The insert operations in this workload were the most time-consuming, but due to the configuration of the workload, this was not affected by the probability p. Therefore it could be related to the higher performance of the scan operation, which makes up a large proportion of the workload.

5.2.3. Varying Memory Distribution

The main memory allocation between the cache and the buffer manager is varied in this experiment. Figure 5.16 shows detailed information about the throughput for different compositions. The y-axis specifies the throughput in thousand operations per second, and the x-axis displays the combinations of available memory space on which we group the workloads. In each group of x:y, x represents the proportion of available memory allocated to the cache, while y indicates the proportion of allocated buffer size. The buffer size was selected to allow all records to fit into the buffer when the cache size was zero. The system's main memory availability is consistently limited to about 0.33 GB, while the other parameters are configured as displayed in Table 5.5. We added the elements starting from the beginning of the array, where most of the requested indices will be located. This method helps to mitigate the deficiencies of the simple cache eviction strategy. The benchmark is executed on a single thread.

Record Count	Operation Count	Distribution	Coefficient
10,000,000	20,000,000	Geometric	0.001

Table 5.5.: Parameters for the benchmark varying the memory distribution

Entity/Distribution	0:1	1:7	2:6	1:1	6:2	7:1
Cache [kB]	0	32,768	81,920	163,840	245,760	294,912
Buffer [kB]	327,680	294,912	245,760	163,840	81,920	32,768

Table 5.6.: Resulting main memory shares for different memory distributions in kB

Based on the previous experiments' cache performance, we anticipate improved

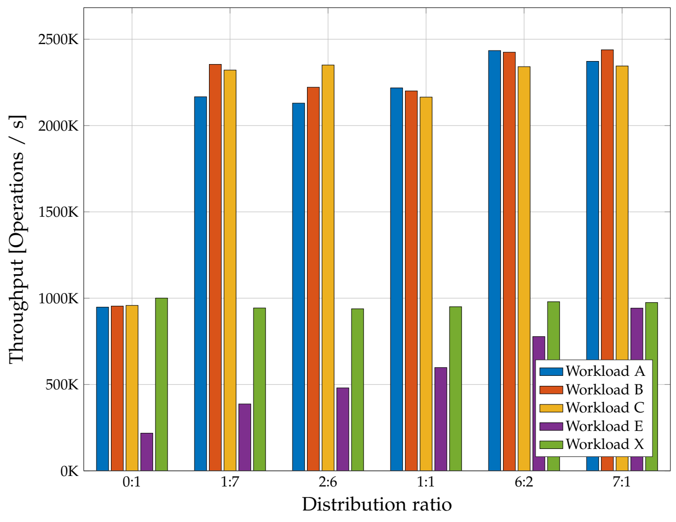

Figure 5.16.: Throughput for workload A for different memory proportions of cache and buffer

throughput for workloads A-C with memory allocated for the cache. The performance is expected to be relatively equal for workloads E and X. Additionally, as we use a geometric distribution, even smaller cache size allocations should yield good performance.

Figure 5.16 presents two notable observations. Firstly, the performance of the cache is relatively constant across different sizes. Workload A-C performs similarly with throughput between 2.17 and 2.43 million operations per second, between the ratios of 1:7 and 6:2. In these configurations, the system performs similarly to the previous benchmarks, where read-heavy and update-heavy workloads have significantly higher throughput. Secondly, a solution with a cache outperforms those without for workloads A, B, C, and E. The difference is most significant for workload A-C, showing more than twice as much throughput. The cache size does not enhance the performance for workload X, while for workload E, more cache capacity relates to better performance.

The high throughput of the cache was expected even with a smaller memory share due to the skewed distribution that keeps some elements hot while not touching others. Additionally, the cache only references and marks the outer nodes of the tree, so fewer leaves are swapped for inner nodes during traversal. This allows the cache to perform reasonably, even with a small buffer size. Workload X does not change with an increase in the cache due to the more costly lookup process involving false positives in the cache for previously deleted elements, as discussed in Section 5.2.1.

5.3. Discussion

Depending on the workload, the cache can provide a suitable solution to increase the overall system performance. In particular, the performance of read and update operations can be improved by using the cache. Figure 5.16 confirms this, which shows that the throughput increases even for different main memory shares between cache and buffer. This finding is further supported by Figures 5.1, 5.3, and 5.5. On the other hand, using the cached solution does not offer performance benefits but creates overhead for insert operations. This is because no lookups on the cache are performed, and the cache needs to be maintained. This is shown in Figure 5.7. Moreover, delete operations slow down due to the increased overhead of maintaining the cache and additionally hamper other operations. In its functionality, the cache can't ensure that an element does not exist, leading to additional lookups in the B+ tree on top of operations in the radix tree. However, depending on the use case, this is unlikely in a real-world scenario, especially when dealing with primary and foreign keys, as in a foreign key table search, we usually do not look for deleted elements.

Entity/Records	200	2000	20,000	200,000	2,000,000
Cache [B]	20,992	212,288	2,118,416	21,173,760	211,737,472
Buffer [B]	4096	65,536	647,168	6,500,352	65,015,808

Table 5.7.: Main memory sizes for different numbers of records for the cache and the buffer manager in byte

The access pattern for the indices connected to the locality of elements in the cache is also crucial for performance. This is evident from Figure 5.14, where the insertion order and limited cache size coupled with a simple eviction strategy meant frequently used elements were not initially in the cache. There was a notable increase in performance as the locality of indices improved, with even the simple eviction strategy being sufficient. As the locality or the eviction strategy worsens, so does the cache's performance. In extreme cases where the indices are uniformly distributed, the cache does not contribute any speedup but introduces additional overhead. The significance of the distribution is even more evident given the main memory consumption of the cache, which is shown in Table 5.7. Although the buffer- and cache size increase linearly with the increase of records, the radix tree takes up approximately 3.3 times more memory than the B+ tree. As a result, the number of elements that can be stored in the cache is limited. Nevertheless, the cache still increases the overall throughput when there is a reasonably small number of hot entries, as depicted in Figure 5.16.

6. Conclusion

This thesis investigated the implementation and performance implications of a cache system that combines features of an in-memory and a disk-based database system, highlighting the importance of efficient index structure design and usage. Initially, we investigated the architecture and mechanisms utilized in disk-based databases. We focused primarily on memory management strategies like files, pages, and buffer management. Next, we explored the B+ tree and linked it to the previous concepts.

We then transitioned to in-memory databases, outlining developments in this area. A detailed discussion of index structures, with a central focus on the adaptive radix tree from [13], provides a contrasting approach to the previously discussed disk-based systems.

The primary objective of this thesis was to implement and evaluate a cache system that utilizes a radix tree as its main-memory structure coupled with a disk-based design. We explained the system's main components: the storage manager, buffer manager, B+-, and radix tree. Examining the system's architecture closely allowed us to understand its behavior during insert, read, update, scan, and delete operations.

To test and evaluate the cache system, we designed an experimental setup. The system's behavior was analyzed in various scenarios by adjusting parameters and collecting performance data.

As a result of these experiments, we discovered several important insights. Our findings indicate that the cache system performs well for read, update, and scan operations but has limitations for insert and delete operations, so its benefits should be evaluated based on the specific requirements of the use case. Additionally, we observed that the index distribution heavily influences the system's performance, with the cache performing better under skewed loads.

In the future, several areas can be further explored. Adjusting the system to function with variable key and value sizes could bring it closer to practical use cases. In addition, eviction algorithms significantly impact performance, as we have observed. Thus investigating different approaches may be beneficial.

List of Figures

List of Tables

Bibliography

[1] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. "System R: relational approach to database management." In: *ACM Trans. Database Syst.* 1 (1976), pp. 97–137.

[2] E. Azar and M. Alebicto. *Swift Data Structure and Algorithms*. Packt Publishing, 2016. ISBN: 9781785884658.

[3] R. Bayer and E. McCreight. "Organization and Maintenance of Large Ordered Indices." In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET '70. Houston, Texas: Association for Computing Machinery, 1970, pp. 107–141. ISBN: 9781450379410. DOI: `10.1145/1734663.1734671`.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking Cloud Serving Systems with YCSB." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: `10.1145/1807128.1807152`.

[5] D. DeWitt, R. H. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. *Implementation Techniques for Main Memory Database Systems*. Tech. rep. UCB/ERL M84/5. EECS Department, University of California, Berkeley, Jan. 1984.

[6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. "Hekaton: SQL Server's Memory-Optimized OLTP Engine." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 1243–1254. ISBN: 9781450320375. DOI: `10.1145/2463676.2463710`.

[7] F. Faerber, A. Kemper, P.-Å. Larson, J. Levandoski, T. Neumann, and A. Pavlo. "Main Memory Database Systems." In: *Foundations and Trends in Databases* 8 (Jan. 2017), pp. 1–130. DOI: `10.1561/1900000058`.

[8] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.).* Pearson Education, 2009, pp. I–XXVI, 1–1203. ISBN: 978-0-13-187325-4.

[9] T. Härder and E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Jan. 2001. ISBN: 3-540-42133-5.

[10] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. "Architecture of a Database System." In: *Found. Trends Databases* 1.2 (Feb. 2007), pp. 141–259. ISSN: 1931-7883. DOI: `10.1561/1900000002`.

[11] T. Lahiri, M.-A. Neimat, and S. Folkman. "Oracle TimesTen: An In-Memory Database for Enterprise Applications." In: *IEEE Data Eng. Bull.* 36 (2013), pp. 6–13.

[12] T. J. Lehman and M. J. Carey. "A Study of Index Structures for a Main Memory Database Management System." In: *High Performance Transaction Systems Workshop*. 1986.

[13] V. Leis, A. Kemper, and T. Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, pp. 38–49. DOI: `10.1109/ICDE.2013.6544812`.

[14] T. Neumann and M. J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance." In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[15] J. Rao and K. A. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory." In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 78–89. ISBN: 1558606157.

[16] B. Schlegel, R. Gemulla, and W. Lehner. "K-Ary Search on Modern Processors." In: *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. DaMoN '09. Providence, Rhode Island: Association for Computing Machinery, 2009, pp. 52–60. ISBN: 9781605587011. DOI: `10.1145/1565694.1565705`.

[17] V. Sikka, F. Färber, A. Goel, and W. Lehner. "SAP HANA: The Evolution from a Modern Main-Memory Data Platform to an Enterprise Application Platform." In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1184–1185. ISSN: 2150-8097. DOI: `10.14778/2536222.2536251`.

Appendix

A. Appendix

A.0.1. Multithreading

The performance of the multithreaded implementation is illustrated in Figure A.1. To avoid race conditions in the buffer, we created a lock for the hash map and individual locks for each frame. Furthermore, we lock all nodes in the radix- and B+ trees from the top down. Additionally, an extra lock is implemented for the root nodes. The highest throughput of the system is achieved with 20 threads. However, it is still slower than the single-threaded implementation due to the added overhead of the locking mechanisms, as we could not optimize it to achieve greater performance.

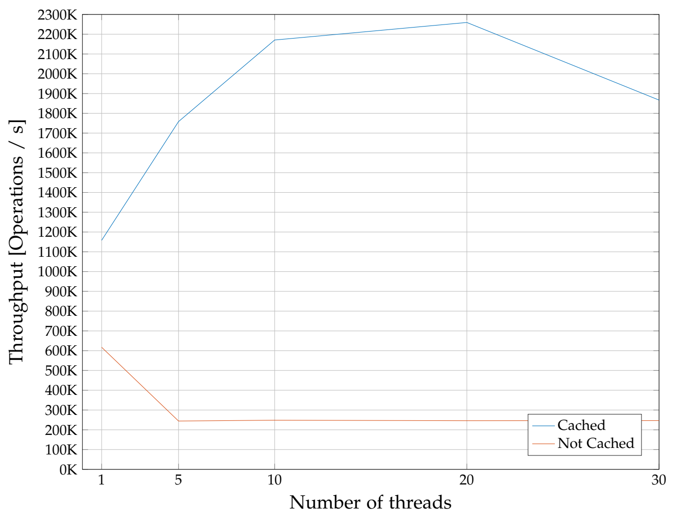

Figure A.1.: Multithreaded performance for workload A