Dynamic Storage Allocation for Concurrently Readable Memory Mapped Databases

Ilias Stamatis

Bachelor's Thesis
Department of Informatics
Alexander Technological Educational Institute of Thessaloniki

June 5, 2018

Abstract

Dynamic storage allocators are a critical component of database systems since they largely affect their performance and safety. Persistent storage allocators used by these systems are different from main memory allocators. Additional requirements such as crash resilience, concurrency control and minimized disk accesses must be satisfied. This creates a demand for efficient, fast and safe allocation of storage space in such environments.

This thesis presents a theoretical design of a dynamic storage allocator for a new concurrently readable memory mapped key value database system. We initially introduce and analyze the internals of the database specifying the requirements of the allocator. Then, we perform a literature survey on dynamic memory allocation concepts, techniques and low-level mechanisms. Finally, we introduce our design proposal. Both in-memory and on-disk structures are presented along with the mechanisms that make the allocator fast, safe and resilient to system crashes.

Foreword

During the summer of 2017 I was working on an administrative framework for 389 Directory Server - an enterprise-class LDAP server - as a Google Summer of Code intern. There I had the chance to collaborate closely with William Brown, my project mentor, who introduced me to the project and the community. A few months later, while I was researching an appropriate thesis topic for my Bachelor's degree, William mentioned that the 389 project was considering alternate key-value storage backends in order to move away from the current Berkeley DB backend. After investigating other existing implementations we realized that they didn't fulfill all our needs so we begun considering developing our own key-value database.

I was then assigned with the task of implementing the extent allocator for this new database system. Since I had to do a lot of research and look at existing implementations I decided to base my thesis on this work. It has been a great experience motivating me to continue studying database internals in the future.

I can't emphasize enough how grateful I am for having the chance to have William Brown as a mentor. William not only helped me by answering hundreds of questions and teaching me engineering best practices, but he also motivated me and put me on the right track on my journey to become a software engineer. This thesis would not be here without his help.

Additionally, I want to thank my university professor Nikolaos Psarras for supervising this thesis and providing me with practical knowledge throughout the duration of my undergraduate studies.

Finally, I would like to thank my parents for their unconditional and endless support not only during my university studies but in every moment of my life.

Ilias Stamatis
Thessaloniki, May 31, 2018

Contents

List of Figures

Chapter 1

Introduction

Dynamic storage allocation has been a fundamental part of computer systems since the beginning of the computer era and a heavily researched topic in academic groups. Many general purpose allocators have been developed over the years for managing main memory such as those implementing the C malloc interface.

However, allocating persistent storage is different than allocating main memory, and when specific requirements or limitations exist custom solutions have to be developed. This thesis focuses on the design of a dynamic storage allocator for a new key-value database. The state of such an allocator must be preserved across different system sessions in contrast to RAM allocators.

This chapter introduces key-value databases, explains the need for efficient dynamic storage allocation, discusses some of the allocator requirements and defines the objective of this thesis.

1.1 Overview of key value stores

Key-value stores are one of the simplest forms of database. They implement an associative array by mapping keys to their corresponding values. Key-value stores are the foundation of almost every complex database from set based, text storage to relational database management systems.

The minimum interface that a key-value database has to provide to a consumer is the following:

- **get(key)**: For retrieving the data that has been previously stored under the identifier "key".
- **set(key, value)**: For storing "value" in the database under the identifier "key".
- **delete(key)**: For deleting the data that has been previously stored under the identifier "key".

Key-value databases are generally highly optimized for reading data. In contrast to the better known relational databases there is no concept of relationships between data nor that of a database schema. Key-value databases fall in the category of the NoSQL databases and the SQL query language cannot be used to retrieve information from them.

Due to their simple nature, search filters cannot be applied to data sets. In case the search key is not known, an iteration over all the keys stored in the database is required which is a slow operation.

Key-value stores can be in memory only and used as non-persistent cache. However, when data persistency is required they need to store their data on disk. Some popular key-value store implementations include LMDB, BerkeleyDB, LevelDB, MemcacheDB and Redis.

1.2 Dynamic storage allocation

Dynamic storage allocation refers to manual management of persistent storage space or main memory. The purpose of a dynamic storage allocator is to provide storage space dynamically to an application at run time whenever the application requests it. When the program is done using that storage space it must inform the allocator which then has to return that space to its free space pool and mark it as available for future reuse.

A great deal of research on the topic exist and many allocator implementations have been developed over the years. A paper from Wilson, Johnstone, Neely and Boles contain an extensive review of past dynamic memory allocation literature between the years of 1961 and 1995 [30]. It presents and compares models, strategies, policies and mechanisms of existing allocators. The survey present by Wilson et al forms the basis of this thesis that we will build upon.

Storage allocation can be over main memory or over persistent storage such as disk. Allocating over disk is different since disk accesses are much more expensive

than memory accesses and reducing the number of disk accesses during allocation and deallocation of space is a requirement in such case. Iyengar et. al have already developed efficient strategies for allocating space over persistent storage, but they focus on allocating over raw disk and they only consider free lists for tracking free space [16].

In contrast to this, in this work we focus on allocating blocks within a single sparse file. The file stores the contents of a key-value database which is solely consisted of this file only. The difference with directly managing disk space lies on the fact that the filesystem sits in between. This means that in many cases the operating system's cache and filesystem implementation may interact with the database system. Understanding these interactions is an important aspect of this design.

The database file is divided into extents. An extent is a logical unit of database storage space allocation made up of a number of contiguous data blocks. The allocator needs to manage the space inside the database file and satisfy allocation requests when write transactions need new extents for storing database data. Hence, the allocator's speed directly affects the write performance of the database making it a critical component of the whole system. It is also the allocator's responsibility to grow or shrink the database file accordingly when needed.

Something that differentiates a database allocator from a general purpose memory allocator, is that its internal data structures and state must not become corrupt in the event of unexpected system failures or crashes. The allocator must suffer no loss of information in such cases and a fast startup time from a cold state is also important. Additionally, certain characteristics of the database impose further requirements to the allocator.

1.3 Objective of thesis

The objective of this thesis is to develop a design for a storage allocator that will be implemented as a part of a new key-value database.

All of the following requirements must be satisfied:

- The allocator needs to be reasonably fast since its performance directly affects the overall performance of the database system. It must be able to satisfy allocation and deallocation requests in time at least logarithmic to the number of the free extents available to the system.

- The allocator needs to be crash resilient and the database image must be consistent at all times. The allocator's internal structures must remain intact in the event of unexpected system crashes.
- Quick recovery from system crashes is required.
- Number of disk accesses for satisfying allocation and free requests should be minimized.
- Unused database space must be deallocated from the filesystem.
- The allocator must never reclaim live storage space. If a database transaction tries to free storage space that is accessed by another transaction, this action must be deferred.
- Due to the concurrent readability characteristic of the database the allocator needs to support a serialized extent allocation mechanism and a parallel and asynchronous extent free mechanism.

1.4 Assumptions

Linux is the initial target platform of the key-value database and because of this reason Linux-specific system calls are mentioned throughout the text. However, most other operating systems and environments support functions similar to the ones referenced here.

Because the database file will be memory mapped a 64-bit environment is assumed. Modern 64-bit CPU architectures such as x86_64 and ARMv8 support 48-bits of virtual address space allowing the memory map and consequently the underlying file to be as large as 256TB.

On the other hand, 32-bit systems are not a realistic option for running a memory mapped database. The virtual address space of each process in 32-bit systems is 4GB part of which is also normally used for mapping the OS kernel. 4GB or less is not a practical size limit for a database and this is why 64-bit systems are targeted.

Finally, the underlying file system is assumed to support sparse files. Sparse files ensure that unused database space can be deallocated from the filesystem and disk when needed. XFS, ext4, Btrfs, ZFS and NTFS all support sparse files.

1.5 Thesis structure

This thesis is organized as follows. Chapter 2 discusses the new key value database system that will be developed and explains some of its characteristics and internal design. Chapter 3 is an overview of memory allocation concepts and low level mechanisms. Chapter 4 presents our proposed design for the extent allocator of the database. Chapter 5 concludes the thesis and proposes future work directions.

Chapter 2

Key Value Database Design

In this chapter the database system for which the storage allocator design was developed is studied. Studying the internal design of the database system is very important since this is what defines the allocator's requirements but also imposes certain limitations to it.

Initially, the internal organization of the data is presented along with the high level overview of the different database levels. Then the concepts of memory mapping and concurrent readability are introduced and discussed. Finally, low level mechanisms such as copy on write and superblock swapping are explained along with the problems they solve.

2.1 Internal organization of data

Multiple tables can exist in the key-value database. Each table internally is stored as a B+ tree supporting a specific type mapping. For example, there could be an "int: int" tree, mapping integer keys to integer values and an "int: char *" tree, mapping integer keys to string values.

A B+ tree is a self-balancing tree having a variable number of children per node. Each internal tree node contains only keys (instead of key-value pairs) and pointers to child nodes. Values are only stored in leaf nodes and the leaf nodes are linked together forming an ordered linked list. An illustration of a B+ tree is shown at Figure 2.1.

All tree nodes have the same fixed size. When a value is too large to fit in a leaf node, it is stored in another extent and a pointer to this extent is stored as

the value in the key value pair. For example, a 7MB jpeg picture could not fit in the leaf node of any database tree and it would require special handling separate to the tree management.

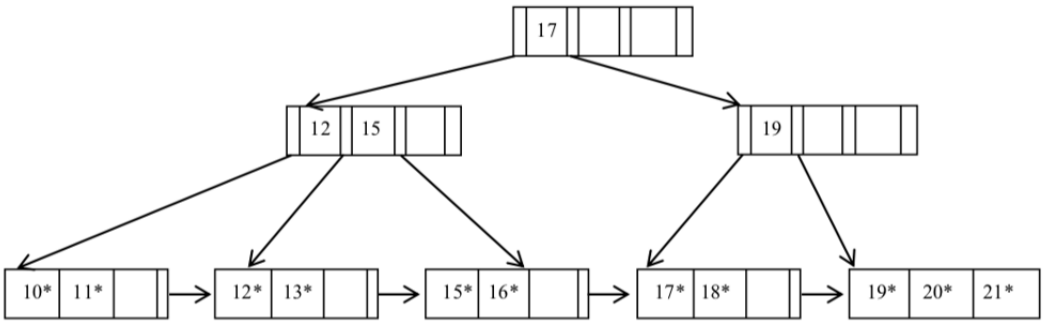

Figure 2.1: A B+ tree. Values are stored in the leaves and leaf nodes form an ordered linked list. Asterisks next to keys represent the presence of a value in the node.

B+ trees index data by key and offer fast lookups of O(log n) time complexity, making a good choice for storing the data internally. The only difference of the database's B+ trees with the textbook B+ trees is that in the database leaf nodes are not linked together.

2.2 Database layers

When seen from a high level architecture point of view, the database is consisted of three distinct layers.

The top layer is the "tree layer". This is in the form of a read or write transaction presented to the consumer. From this layer any of the trees that exist in the database can be accessed.

The middle layer is the "transaction layer". This manages the set of transactions in trees and is responsible for cleanup of unused tree nodes and components once transactions end. Only two types of transactions are supported by the database; read transactions also known as "readers" and write transactions also known as "writers"". Transactions are completely isolated.

The lower layer is the "extent layer". It provides storage space for the layers above on request, and frees space when no longer required. Database storage is split into extents. An extent is a logical unit of database storage space allocation

made up of a number of contiguous data blocks. The extent layer is implemented by the storage allocator.

2.3 Memory mapping

The whole database file is mapped into memory. That means that a contiguous segment of the process's virtual memory has been assigned a direct byte-for-byte correlation with the database file. On Linux, a memory mapping can be created using the *mmap* system call.

The concept behind using a memory mapped file is that memory and disk are treated as if they are the same. When data are fetched from the database the read comes directly out of the memory map and there is no need for user space memory buffers, memory copying or memory allocation. This is known as *"zero-copy"* since no data needs to be copied between the kernel and the user space [3][27].

Changes or writes to the memory map are carried through to the underlying file and stored on disk. Again there is no need for write buffers and buffer tuning. As a consequence, there is no need for application level caching either. In such a case the system relies on the operating system's caching mechanisms [3].

The virtual address to which the file is mapped into changes every time. That means that no absolute pointers shall be saved in the database file. Pointers relative to the beginning of the mapping can be used instead. An "address translation mechanism" is needed in order to translate relative pointer to absolute pointers and vice versa.

2.4 Concurrency support

The key-value database can be characterized as *"concurrently readable"*. A concurrently readable system is defined as a system where 0 or 1 writers can act in parallel with multiple readers. At any given time only a single write transaction can be operating in the database. We say that writers are *serialized*.

This has the following consequences for the database system:

- Writers do not block readers.
- Readers do not block writers.
- No deadlocks can ever exist in the database.

- Reads scale linearly with available hardware resources.

The database system can be accessed by a single program or process only, so no multi-processing is involved. Concurrency comes from multiple threads executing in parallel within the same process.

Writers being serialized means that no multithreading support is required from the extent allocator, since only write transactions modify the allocator's structures in a way that requires locking. This will be further discussed in Chapter 4.

2.5 Copy on write

Copy-on-write is a resource management mechanism used to efficiently implement a copy operation on modifiable data structures. In the case of the database system, all transactions must be isolated operating on their own personal "snapshot" of the database. This essentially means that each transaction must maintain their own copies of the database trees. However, making a deep copy of all database trees for every new transaction is an expensive and inefficient operation that also consumes excess storage space which may cause out of space conditions for the system.

In contrast, copy-on-write is a technique that helps reduce this overhead. With copy-on-write only modified nodes need to be copied and the rest of the nodes are shared between the trees. This means that the only excess storage consumed is between a former data generation and the current one. At best, only a single tree node may be copied, but in cases of complete data rewrite, the entire tree is copied.

Copy-on-write works as follows. When a new write transaction is initiated, it initially shares the same root node with the previous committed read-only tree. When the new transaction needs to modify some node of the tree, it first copies the data within the node to a new memory location. Modifications are then performed on the new node. This new node is now owned by the new transaction tree and the old node belongs to the previous tree. All other nodes are shared by both trees.

Figure 2.2 can serve as an example for understanding the concept better. Current transaction A, a read transaction, is operating owning the tree whose root node is root 9. A new write transaction B is launched, sharing the same tree as A initially. When B needs to modify leaf 8, this node is first copied into leaf 12 and then modified. Then branch 6 has to be modified too since it contains a pointer to leaf 8 that must be updated, and root 9 has to be modified as well for maintaining

a pointer to branch 6 that has just been altered.

This series of copy and write operations results in a new tree with root 13 as its root node owned by transaction B. The two trees share six nodes between them; nodes 1 to 5 and node 7. The trees have to co-exist as long as both transactions are alive. When transaction A terminates, nodes 9, 6 and 8 can be reclaimed as long as there are not other references to them.

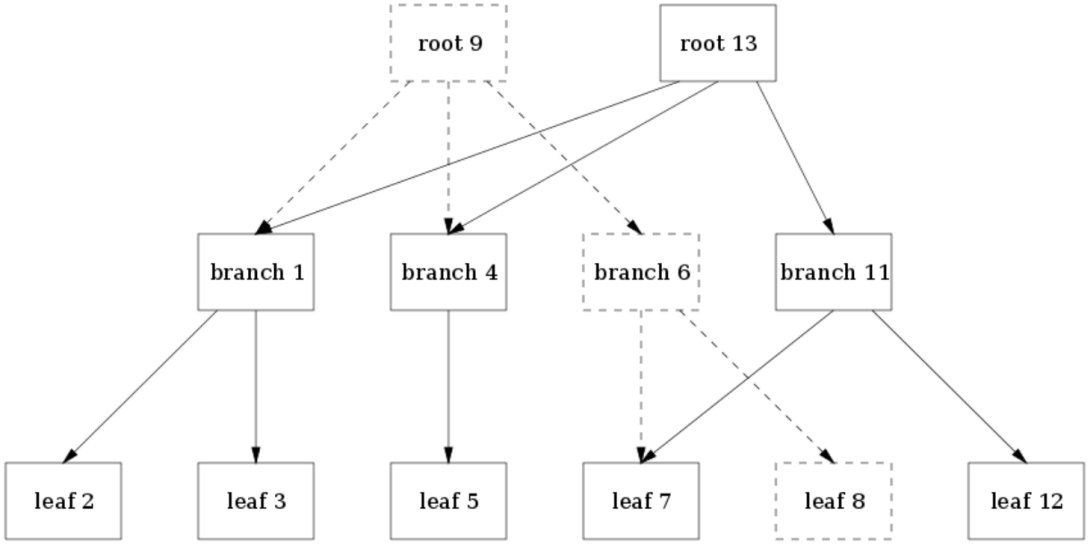

Figure 2.2: Illustration of the copy-on-write technique. Read transaction A owns the root 9 tree and write transaction B owns the root 13 tree. Various nodes are shared between the two trees.

An important observation is that every leaf node modification is propagated up to the root node and all the nodes in between have to be updated as well. Changes being propagated is the reason for which the leaf nodes of the database's B+ trees are not linked together in contrast to the textbook B+ tree definition. If that was the case, many more nodes would need to be copied in every tree modification. For example, a modification of the rightmost leaf node would result in the entire tree being copied.

2.6 Crash resilience

Crashes may happen at any time causing unexpected termination of the database system. Crashes happen because of application bugs, operating system bugs, hard-

ware failures or simply because of sudden loss of power. Nevertheless, the database image must remain consistent at all times.

In case of an unexpected system failure the database file should not become corrupted. This affects the overall design of the allocator since it has to ensure that its internal structures are always valid and consistent[1]. If the internal structures become corrupted that would lead to "orphaned" extents. These extents of which the allocator lost track would be hard to reclaim and reuse without a full scan of the database content. Additionally, allocated extents could potentially appear as free risking data to be overwritten.

The database system must be able to quickly recover from crashes. Copy-on-write helps prevent corruption of the database data trees since live data are never overwritten when using it. Even if the database system crashes in the middle of a write operation, a consistent copy of the tree of the last committed transaction will be present and will be used after the system comes up again. This removes the need for write-ahead logging and transaction log cleanup maintenance.

2.7 Superblock swapping

Pointers to the root nodes of the last consistent tree images are stored both in memory and on disk in a special block called the *superblock*. The superblock is a database block containing various metadata about the database such as its size, various flags and the above mentioned pointers. Upon system startup the superblock is examined and the position of the root nodes of the database trees is determined. When a transaction modifies a tree it results in a new root node being created which must be then stored in the superblock.

Updates to the superblock cannot be in place since that would risk the database's consistency if a crash occurred during a superblock update. The superblock must be copy-on-write too in order to prevent that from happening. For this reason certain extents at fixed locations are reserved and used to store superblock copies. When the superblock has to be modified, the contents of the current superblock are copied into one of the reserved extents[2] and changes are made there.

Additionally, the superblock contains an integer field called *priority*. When

[1]This is also another aspect that differentiates the database's extent allocator from general-purpose memory allocators which do not care about crash resilience.

[2]It can by any extent except for the one holding the current superblock.

a transaction that has updated the superblock commits[3], the priority field must be incremented by one. The priority must always be the last field to be updated in the superblock. When the system is started again, it examines all superblocks stored at the reserved extents and picks the one with the highest priority. This superblock contains pointers to the last valid and consistent tree images.

If a transaction didn't update the superblock and change its priority before a crash occurred it is like it never really happened. After a system restart, any changes made by that transaction will not be visible since the last valid superblock will be used containing pointers to previous consistent images of the database trees. The very action of committing a transaction is therefore equivalent to updating the superblock's priority field[4].

Prior to updating the priority field an *msync* call must be issued on the memory mapped file. The *msync* system call flushes any changes made in memory back to the filesystem thus synchronizing the file with the memory map. To ensure that a transaction committed properly two *msync* calls are required. One just before updating the priority field and one immediately after updating it. Otherwise memory pages may be flushed to disk in random order risking the consistency of the database file. A commit can be considered persistent and complete only after the second *msync* call succeeds.

An important consequence of the copy-on-write and *superblock swapping* mechanisms is that there is no need for any special recovery procedures after a system crash. Upon startup the database executes the exact same series of steps independently of whether the system shut down normally or not.

2.8 Allocations before liberations

Due to the copy-on-write mechanism a certain allocation pattern is observed. Each write transaction modifies one or more leaf nodes and their ancestors by first copying them into new extents. That means that each write transaction must allocate a certain number of extents and free the same number of same sized extents or less in case some of them are referenced by other transactions.

[3]Each and every write transaction has to update the superblock since changes to any database tree result in a new root node.

[4]A write transaction to be accurate, since readers do not need to modify the trees nor the superblock at all.

An important question arises that if the order of these allocations and deallocations matters. It turns out that the order here is very important because in a copy-on-write system extents freed during a transaction can only by allocated for use in a subsequent transaction [20, Lemma 1 on p. 4]. Hence, either this logic must be implemented in the allocator or a simple convention can be used instead.

The convention to avoid these issues is that write transactions must allocate all extents they will utilize first. When the write phase is complete but the commit not completed, unused extents will be freed. By following this simple convention the allocator does not need to implement extra logic for preventing freed extents from being allocated in the same transaction.

Chapter 3

Memory Allocation Concepts and Mechanisms

This chapter provides the reader with the necessary background on dynamic memory allocation. We initially introduce relevant terminology and concepts such as memory fragmentation and ways to avoid it. Then we discuss the low-level mechanisms used by allocators in order to keep track of free space and satisfy allocation requests.

Although discussions on memory allocation normally refer to main memory, all concepts and mechanisms described in this chapter can be applied to allocation over persistent storage devices such as disks as well. For this reason "memory" and "storage" are used interchangeably for the most part of this chapter.

3.1 Dynamic memory allocation

With dynamic memory allocation an application can request new memory for its purposes while operating and free it at any time or any order. This comes in contrast with static memory allocation where all memory that is going to be used by the program is known at compile time and fixed until the program terminates.

Underneath, the memory space is managed by the allocator and is divided into smaller blocks or chunks of variable sizes. A block is consisted of a number of contiguous words of memory (or file system blocks or disk sectors depending on the case). A memory word on most modern machine is four 8-bit bytes (32 bits) or eight 8-bit bytes (64 bits) depending on the CPU architecture. Each block of

memory can then be in two distinct states: allocated (in use) or free (available for allocation). Figure 3.1 illustrates that.

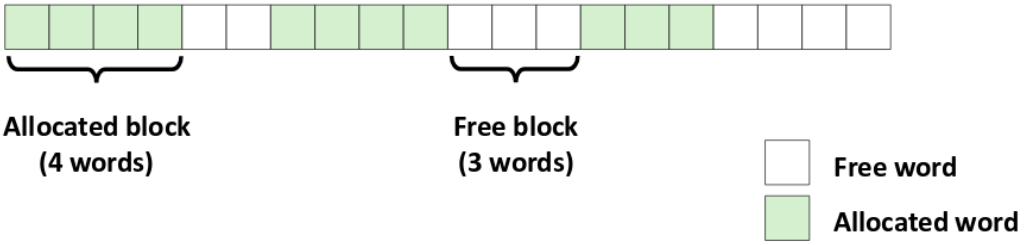

Figure 3.1: Memory blocks made up from a number of contiguous words. Each rectangle represents a memory word. Sequences of white rectangles represent free blocks and sequences of green rectangles represent allocated blocks.

For performance reasons memory blocks are aligned on word boundaries and normally partial words are not allocated. That means that requests for non-integral numbers of words are usually rounded up to some word or other multiple [30, section 3.2]. Allocators need to keep track of which blocks are free and which are not and various data structures can be used for this kind of bookkeeping.

The memory area managed by general-purpose memory allocators is called *heap*[1]. When the allocator needs more memory it can request it from the operating system via system calls. That means that the heap's size also changes dynamically and the heap can grow or shrink while the program is running.

3.2 Memory fragmentation

Fragmentation is generally defined as the inability to use memory that is free and is a source of wasted memory in the allocator [14, section 2.4]. Traditionally it is divided into internal and external fragmentation.

3.2.1 Internal fragmentation

Internal fragmentation is defined as wasted memory inside an allocated block. This happens because more memory is allocated than required in certain circumstances.

[1]Not to be confused with the heap data structure.

For example, if an allocator is only allocating chunks of memory that are multiples of 4 bytes and the program makes an allocation request of 13 bytes, it will get back a chunk of memory that is at least 16 bytes. This remainder is wasted causing internal fragmentation. [30, pp. 8-9]

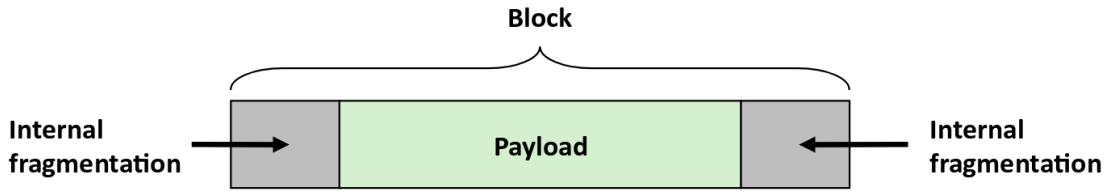

Figure 3.2: Illustration of internal memory fragmentation

3.2.2 External fragmentation

External fragmentation arises when free blocks of memory are available for allocation, but are too small to hold objects of the sizes actually requested by the program. That practically means that while there is enough free memory in total to satisfy an allocation request, this memory is not contiguous thus it cannot be used to satisfy the request. [30, pp. 8-9] Figure 3.3 illustrates that.

Figure 3.3: Illustration of external memory fragmentation. After the free request in step 4 there is a total of 32KB free space. However this space is not contiguous causing an allocation request of 32KB to fail.

3.3 Combating fragmentation

Fragmentation is one of the main problems that a memory allocator has to deal with. However, a study from Johnstone has indicated that well-known allocation policies suffer from almost no true fragmentation when implemented properly [17]. Memory fragmentation is combated thanks to certain techniques some of which are presented below.

3.3.1 Splitting large blocks

When an allocation request is made, it is possible that all available memory blocks are much larger than the size requested by the application. What is possible in this case is to split some large block into two or more "pieces" one of which having the requested size. This memory block can then be used to satisfy the allocation request and the remainders are returned in the free space pool. Splitting large blocks is a way to combat internal fragmentation and decrease wasted space.

3.3.2 Coalescing adjacent free blocks

When two or more adjacent memory blocks are free they can be merged together by the allocator forming a single large free memory block. Coalescing blocks is the opposite of splitting a large block into smaller ones and it can be used in order to reduce external memory fragmentation.

Coalescing can be attempted immediately when a block of memory is freed which is known as *immediate coalescing* or be deferred to some point in the future in which case it is called *deferred coalescing*.

The rationale behind deferred coalescing is that an allocation request for the same size that just got freed may follow very soon or even immediately. By deferring it for later we avoid wasting time on coalescing two blocks and then shortly splitting them again. Coalescing can be performed periodically or deferred until all memory runs out. Proper scheduling of coalescing requires a great deal of research and analysis which is beyond the scope of this document.

3.3.3 Compacting memory

Another technique known as *compacting memory*, moves all allocated blocks into consecutive locations. As a result, free memory blocks come together and then

can be coalesced reducing or even eliminating external fragmentation.

Memory compaction is not normally used by general-purpose memory allocators since that would invalidate all application pointers and it would require very special handling. However, memory compaction is accessible when requested from filesystems that implement the correct operations. This is commonly referred to as *defragmentation*.

3.4 Keeping track of free space

Allocators internally need a set of data structures to keep track of locations and sizes of free blocks. When an allocation requests is issued, the allocator searches its internal structures in order to find a memory block suitable to satisfy the specific request. Normally, each block of memory has some metadata associated with it.

3.4.1 Block headers

Most allocators use a header field within each memory block to store metadata specific to the block. This metadata most commonly includes the block size. Apart from this it can include other useful information about the block such as whether it is allocated or free, where the next free block is located etc.

This header field is not directly visible to the user. As Figure 3.4 illustrates the address returned to the user is the one right after the end of the block header. Because the size of the block header is fixed, we can go back from that address to the start of the block header by doing a simple subtraction. So when the user wants to free a memory block they do not need to explicitly provide us with the block size. By doing simple pointer arithmetic we can examine the block header and see how large is the block that we need to free.

Additionally to block headers, sometimes footers may be contained in memory blocks as well. This helps in the process of coalescing where the allocation status of the free block's neighbors must be checked. Checking the next block is easy. We know where the next block's header begins, because this is where our free block ends and we already know our free block's size. Checking the previous block is more challenging since we don't know its size, hence we don't know where its header is located.

For this purpose Knuth introduced the concept of *boundary tags* [22]. Each block of memory additionally to its header may also have a footer field known as

boundary tag, repeating information about its size and its allocation status. We can then use the previous block's footer to check whether it is free and jump back to its header if needed knowing its size.

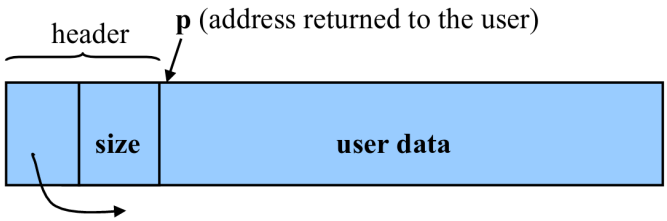

Figure 3.4: An illustration of the "hidden" header field and the actual user memory contents that constitute a memory block. The addressed returned to the user is p and not the starting address of the header.

Block headers and footers impose an overhead since they consume some portion of the available memory, so it is important to keep them as small as possible for general purpose allocators. Standish has introduced an optimization for reducing the boundary tag overhead [28].

3.4.2 Link fields within block headers

Usually one or more fields inside the block header are used to point to other free or allocated blocks of memory. A common technique is to have the free blocks form a single linked list. This way the list nodes are embedded in the free block themselves. Such a list is called a *freelist*. When a block is allocated, it is removed from the freelist and returned to the user. Later, when this block is freed, it is put back on the freelist so it can be re-used to satisfy new requests. An illustration of a freelist can be seen in Figure 3.5. Link fields can be used in order to form other linked data structures too.

3.5 Mechanisms and policies

We will now present a relatively conventional taxonomy of allocators based mostly on the mechanisms and data structures used for their implementation. Additionally, certain allocation policies will be explained.

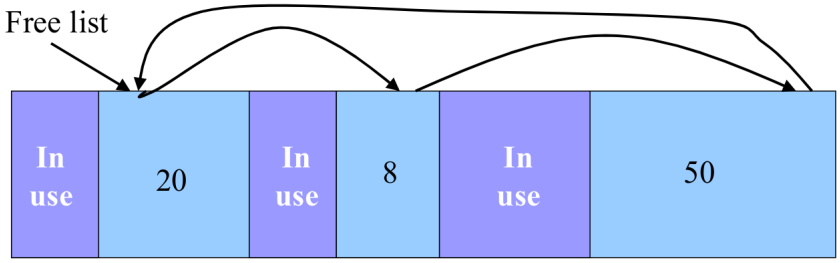

Figure 3.5: Link fields within block headers form a circular singly-linked list.

3.5.1 Sequential fits

Several classic allocator algorithms are implemented using a single linked list of all free blocks similar to the one we discussed above. This list might be singly-linked or doubly-linked. In order to find a free block, the freelist is searched with some algorithm or allocation policy which also defines the variant of the so-called *sequential fit*. The three more popular variants are best-fit, first-fit and next-fit while there are more that are less common.

3.5.1.1 Best-fit

A sequential best-fit allocator has to traverse the whole linked list starting from the beginning, searching for the smallest available free block large enough to satisfy the allocation request. It can stop earlier if a perfect fit is encountered. This policy can be used to minimize internal fragmentation of blocks. However it is slow since it takes linear time to traverse the whole list in the normal case where a perfect fit is not available [30, p. 30].

3.5.1.2 First-fit

A sequential first-fit allocator searches the freelist from the beginning and uses the first block large enough to satisfy the allocation request. If the block is much larger it can be split into two blocks and the remainder is put back to the freelist. The advantage over best-fit is that normally there is no need to traverse the whole list. A problem that arises though, is that many small blocks tend to accumulate near the beginning of the list as a result of splitting larger blocks. These small blocks can increase search times and make the allocator slow when larger blocks are requested [30, pp. 30-31].

3.5.1.3 Next-fit

Next-fit is just a simple variation of first fit. A pointer is used to track the position in the list where the last search was satisfied. Then, the next search continues from that position. As a result, searches do not always begin from the same place. Next-fit was initially introduced as an optimization of first-fit, however it has been shown to cause more fragmentation than the other sequential fit variants [30, pp. 31-32].

3.5.2 Segregated free lists

The problem with having a single potentially lengthy list is that search times are O(n). For this reason the segregated free lists mechanism is commonly used.

In this technique we have many lists, each one holding blocks of a particular size or a range of sizes called *size class*. Upon allocation, a block is removed from a free list matching the allocation size requested and upon free the freed block is similarly added to a free list of the matching size. The benefit of using separate lists is that for certain sizes, depending on the number of lists we use, we can achieve constant time allocation.

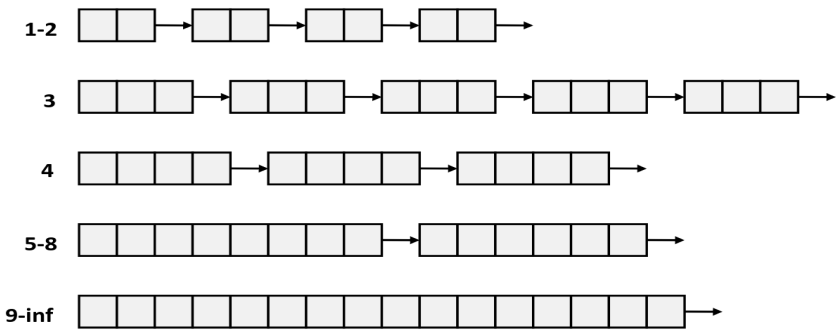

Figure 3.6: Segregated lists illustration

The segregated free lists mechanism probably originates from a paper by Comfort [5]. Both Doug Lea's dlmalloc and GNU libc's malloc implementations use segregated lists which they call bins [23][24].

3.5.3 Tree structures and indexed fits

More sophisticated data structures such as tree structures can be used in order to index blocks by exactly some characteristics of interest such as block size or block memory address. Usually B trees, B+ trees or Red-Black trees are used but any self-balanced tree or other indexed data structure could be used. For example, the "Fast Fits" allocator uses a Cartesian tree [29]. Wilson et al. refer to this kind of mechanism based on indexed structures as *indexed fits* [30, pp. 40-41].

Balanced trees support insert, delete and search operations in O(log n) time. As a result, a data structure indexed by block size can allocate new blocks of any size in logarithmic time and it can be used to implement a best-fit policy. Indexing by memory address or some address offset is useful when we care about storage locality, for example when we want to allocate a block near some other block. Additionally, it makes it easer to search for neighbors when trying to coalesce blocks.

The XFS file system uses two B+ trees to track its free space; one B+ tree that is indexed by block number and another one that is indexed by the size of the free space block [31]. Likewise, the jemalloc allocator - used in FreeBSD's libc and previously in Firefox - makes use of a red-black tree to store metadata about blocks falling into its largest size class. For its smaller size classes it employs the buddy technique which we do not describe in this text [6].

3.5.4 Bitmaps

A bitmap is a simple vector of one-bit flags, with one bit corresponding to each word of the whole memory area or each file system block in the context of a file system block allocator. Those bits are used to indicate the allocation status of the corresponding blocks. Figure 3.7 illustrates this concept.

When the bitmap is large, bitmapped allocators are slow because search times are linear. However, an advantage they have is that they incur much less space overhead in comparison to block headers, thus wasting less memory. Additionally, bitmaps support localized searching since free memory is indexed by address order.

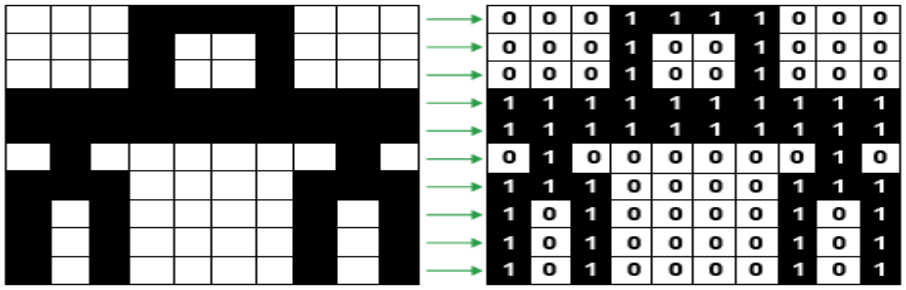

Figure 3.7: Illustration of a bitmap used by a file system's block allocator. Each black rectangle represents an allocated 4K file system block and its corresponding bit is set in the bitmap. White rectangles represent free blocks and their bits are not set in the bitmap. In this case 100 bits (~13 bytes) can be used to describe the allocation status of 400K of file system space.

According to Wilson, bitmapped allocation has never been used in a conventional allocator. Instead it is used or had been used in file system allocators and mark-sweep garbage collectors [30, p. 42].

Chapter 4

Design for a Crash Resilient Extent Allocator

This chapter introduces a theoretical design for the extent allocator of the key-value store. Both the in-memory structures and the on-disk format are described and the allocation and deallocation operations are explained. Additionally, the mechanisms making the allocator crash resilient are presented.

4.1 Extent sizes

Extents are going to be allocated in sizes that are multiples of 4K. This value matches the physical sector size of modern hard disks and solid state drives, ensuring extent alignment on 4K boundaries. The reason for which we want to match the physical sector size is preventing unnecessary disk accesses under certain circumstances. For example, a single 4K extent which is not properly aligned would require two disk fetches instead of one in order to retrieve its content.

It is important to note that the database's trees will also have nodes sizes of about 4K for the same reason[1]. That means minimal or no internal fragmentation at all inside allocated extents. In contrast to general purpose memory allocators, we never expect allocation requests of a few bytes only.

[1]Tree node sizes will actually be a little smaller than 4K allowing space for extent metadata.

4.2 In-memory data structures

In our proposed design, two B+ trees are used for tracking free space usage. One tree is indexed by extent size, which we will refer to as the *Size Tree*. The second tree is indexed by extent address offset relative to the beginning of the file's memory mapped base address. We will refer to this tree as the *Offset Tree*.

Both trees track free extents only. Specifically, keys of the Size Tree are 64-bit unsigned integers representing extent sizes. The values of the tree are 64-bit pointers, where the highest bit is used as a flag indicating whether the free extent is reclaimable or not. We will explain the purpose of this bit later. The lower 63 bits store the extent's address offset.

The Offset Tree can be though as the opposite of the Size Tree. Its keys are 64-bit offsets. While 63 bits or even less would actually be sufficient for storing the offset value, we include the extra bits to achieve data structure alignment. The tree's values are 64-bit extent sizes.

Both of these tree structures are in-memory only and their nodes are not embedded into extent headers[2]. Having the data structures in-memory only allow for faster data structure manipulations, faster searches and reduced number of disk accesses. Another benefit is that on-disk metadata overhead is significantly less reducing the percentage of wasted disk space.

4.2.1 Data structure choice rationale

To explain why we believe that indexed structures such as B+ trees is the best option to keep track of free extents for the database, we compare them with the alternatives presented in Chapter 3.

While bitmap operations can be made fast enough to use when the underlying storage space is not too large, linear search times cannot be avoided. This becomes much worse as the database grows and we can expect the database file to become very large. Additionally, bitmaps track both free and allocated blocks while the indexed tree structures keep track of free extents only. This means both faster

[2]Since the trees are in memory, a node size of 64 bytes can be used in order to match the cache line width. 64 bytes allow for 3 keys and 4 pointers to be stored inside each single tree node, leaving an extra space of 64 bits. That extra space can be used for storing metadata about the node such as a checksum or other.

searches and less space overhead for the trees as the database becomes larger and larger.

When it comes to segregated lists the problem is that we cannot have a different list for each possible extent size. Especially, in the case of a database where an allocation request can be anywhere from a few kilobytes to a few gigabytes or even larger. Thus, we have to divide sizes into different size classes, but then searches inside those lists take linear time. Furthermore, when the list for the size requested is empty we need to iterate over the different list heads until we find one that is not empty. This operation takes linear time as well[3].

In contrast to that, all B+ tree operations including searches take logarithmic time which is considered to be fast enough for our purposes. Furthermore, as the offset tree supports locality searches, this makes coalescing easier without the need for on-disk boundary tags. This is also safer as it eliminates a potential source of corruption and inconsistency as flushing the boundary tags would require its own synchronization mechanisms.

4.3 On-disk extent headers

The only kind of metadata that we need embedded into extents is their sizes. This can precede the actual data forming an extent header of a single field.

Maintaining extent sizes on disk is necessary in order to free extents without requiring the caller of the free routine to explicitly provide us with the extent size. This information cannot be extracted from the in-memory B+ trees, since they only maintain information about free extents and not allocated ones. Saving allocated extents' sizes in extent headers on disk can solve this problem.

The size field has to have the same length as the sizes stored in the in-memory trees i.e. 64 bits long. To get the actual extent size in bytes we need to multiply this value by 4 KB. Thus, the theoretical maximum for an extent size is $2^{64} * 4$ KB = 64 ZB (zettabytes).

[3] A possible optimization is to have a binary search tree maintaining pointers to the different list heads but this is also O(log n) and makes the implementation even more complex.

4.4 Satisfying allocation requests

Whenever an allocation requests comes, the first thing that we need to do is add the extent header size to it (64 bytes) and round up the result to the nearest 4K multiple. This is the actual size of the extent that we need to return back to the caller.

In order to find an extent with such a size we perform a lookup operation on the Size Tree. We can then distinguish from the following cases:

- An extent of the requested size is present on the tree. In that case we use this extent to satisfy the request.
- An extent of the requested size is not present on the tree, but there is an extent of a larger size. In that case we split the large extent into two and satisfy the request with one of the resulting extents.
- There is no extent of a size equal or greater than the requested one. In that case we perform the coalesce operation trying to coalesce adjacent free extents. After this operation is complete the tree is searched again for an extent of a size large enough to satisfy the request.

It is important to note that only extents for which their reclaimable bit is set are considered for allocation. Non-reclaimable extents are completely ignored during this lookup.

If everything else fails we try to grow the database file by at least the size of the requested extent. If this operation succeeds we are able to proceed by performing a tree lookup again as previously explained. In case we are unable to grow the file we need to abort the current write transaction.

4.4.1 Allocating exact fits

Once an exact fit has been encountered in the Size Tree the following actions have to be performed in order to satisfy the allocation request:

a) The key/value pair has to be removed from the Size Tree and the extent offset must be noted.

b) The corresponding key/value pair has to be removed from the Offset Tree as well. The address offset from the previous step is used as the key for the removal operation.

c) The size field on the extent header has to be updated.

Finally, the extent offset plus the header size have to be added to the base memory address where the database file has been mapped by the mmap call, as seen below:

returned address = mmap base address + extent offset + header size

The resulting memory address is returned to the caller.

4.4.2 Splitting larger extents

When free extents matching the requested size do not exist, some larger extent has to be split in order to avoid internal fragmentation. The larger extent is split into two extents. One extent of the requested size which is returned to the caller and the remainder extent which is put back on the trees.

The search operation on b+ trees can work in such a way that if the queried key is not found, the next key can be returned instead. In case of the Size Tree, that means that if a size is not found, an extent of the next larger size available will be returned.

Suppose X is the size requested and Y is the next available size of an extent located at y. The split operation can be implemented as follows:

a) The extent of size Y has to be removed from both trees.

b) The extent of size Y - X located at y + X has to be added to both trees.

c) The size field of the extent located at y has to be updated with the value X.

For example, let's suppose that a 4K extent has been requested but the next available size that has been found in the Size Tree is 12K and its offset from the beginning of the file's memory mapped base address is *0xff000000*. We first remove this extent from both trees. Then we add the key/value pair "8K : *0xff000000* + 4K" i.e. "2 : *0xff001000*" to both trees. The last step is to update the extent header located at *mmap base addr + 0xff000000* with the value 1. The memory address that is finally returned to the user is *mmap base addr + 0xff000000 + 0x40*.

4.4.3 Coalescing extents

Coalescing is deferred until there is no free extent large enough to satisfy an allocation request.

Due to the design of the B+ Tree, an ordered linked list of all Offset Tree nodes is present which allows for efficient coalescing operations. We begin traversing the list starting from the first key of the first node. For each offset value we add its corresponding size to it and check if it matches the next key in the list. If the values match, it implies that two consecutive free extents have been encountered. We continue checking the list to determine if there are further extents adjacent to the original matched pair. We then have to merge the adjacent extents encountered into a single one. This is done by removing the individual key/value pairs from both trees and adding a new one representing the resulting extent.

Two alternatives can be used in the coalescing process. One is to traverse the whole list and coalesce everything possible. The other alternative is to stop as long as one of the resulting new extents is large enough to satisfy the allocation request which triggered the coalescing operation.

Due to the copy-on-write mechanism of the database and the fixed-size tree nodes, allocation requests for the same extent size will probably follow. For this reason it is better to complete the coalescing operation instead of stopping in the middle, in order to avoid starting from the beginning in the next allocation request. This is also the reason for which we prefer deferred coalescing to immediate coalescing. We want to avoid merging extents of sizes that will probably need to be allocated again shortly.

4.4.4 Growing the database file

Growing the database file is deferred until there is no other way to satisfy an allocation request. The database can grow by some fixed amount of space each time e.g. 256 MB. Of course if the allocation request which led to the grow operation is larger than that, even more space has to be allocated from the filesystem.

First, the database file has to be enlarged using the *ftruncate* Linux system call. Then, the memory map needs to be updated in order to reflect the change. This is done using the *mremap* system call. The extra file space needs to be marked as a new large extent and be added to both trees. Further allocation requests can be satisfied by splitting this new extent into smaller pieces.

When the `MREMAP_MAYMOVE` flag is specified for the mremap call, the memory map may be relocated to a new virtual address. Absolute pointers into the old mapping location become invalid and need to be recalculated using offsets relative to the starting address of the mapping [26]. As a consequence, for this recalculation to happen all existing read and write transaction must halt until the operation is complete.

Shrinking the database file requires a similar series of steps, assuming that a contiguous free extent resides at the end of the memory map.

4.5 Deallocating space

Freeing extents from within a transaction requires special care, since we need to be sure that no one else is ever going to access the same storage space again. Additionally, we need to make sure that once space is marked as free inside the database, this space is also reclaimed by the filesystem reducing the size of the database on disk.

4.5.1 Liberation versus reclamation

The database allocator has a major difference in comparison to general purpose memory allocators when it comes to freeing storage space. In a general purpose memory allocator, when some memory region is freed by the user, the allocator is free to reclaim that space and use it for further allocations. However this is not always the case in a concurrently readable database.

It is possible that while a write transaction frees an extent after copying its contents to a different extent due to the copy-on-write mechanism, some older read transaction is still accessing the same extent's contents. That means that the extent's space cannot be reclaimed yet, until all read transactions that reference this extent terminate. Thus, freeing an extent and reclaiming an extent are considered two distinct operations and are treated differently by the allocator.

This distinction is reflected in the Size Tree by the reclaimable bit. When this bit is off, it means that even though the corresponding extent is free in the current snapshot of the database it cannot be allocated to a new transaction yet. When the bit is on, the extent is both free and available for satisfying allocation requests.

4.5.2 The free/reclaim interface

Two different functions have to be offered for freeing and reclaiming extents:

- int free_extent(void *ptr, bool reclaimable) // for writers
- int reclaim_extent(void *ptr) // for readers

The *free_extent* function frees the extent pointed to by *ptr* and marks it as reclaimable or not depending on the value of the *reclaimable* flag. This function must be invoked by writers only. When the extent that is being freed is not referenced by any read transaction, the *reclaimable* flag must be set, otherwise it must be zero.

The *reclaim_extent* function can be used to mark an extent as reclaimable. A call to *free_extent* for the same extent must always precede. The *reclaim_extent* should only be invoked by readers, and specifically upon the termination of the last reader referencing an extent that has been freed before.

4.5.3 Freeing extents

When *free_extent* is invoked, the extent being freed needs to be added to both free space trees. For that to happen, the offset of the extent from the beginning of the database file must be calculated first and its size must be extracted from its header. Additionally, if the *reclaimable* flag is true, the corresponding bit must be set on the Size Tree during insertion.

4.5.4 Reclaiming extents

When an extent is marked as reclaimable by some upper layer using the *reclaim_extent* call, it means that the extent is already present in the trees. The only thing left to do is update the Size Tree by updating its reclaimable bit. From now on, the allocation algorithm is free to use this extent in order to satisfy new allocation requests.

4.5.5 Hole punching free extents

When the underlying filesystem supports sparse files[4], the *fallocate* Linux system call can be used along with the `FALLOC_FL_PUNCH_HOLE` flag in order to deallocate filesystem space anywhere within the file, creating holes. Whole filesystem blocks are removed from the file and subsequent reads from the freed range returns zeros [7]. In case of a memory mapped file, the *madvise* call must be used instead with the `MADV_REMOVE` option set.

[4]XFS, ext4, Btrfs, ZFS, NTFS all support sparse files

This technique, called *hole punching*, can be used to create holes in the database file where free extents reside in order to decrease the space occupied by the database on disk. Additionally, the allocator doesn't need to zero out storage space manually. Free extents are zeroed by the *madvise/fallocate* call ensuring a clean state upon allocations of new extents. Hole punching can be applied only to reclaimable free extents and only after the transaction marking them free commits.

While hole punching results in decreased disk space, it comes with a slight performance overhead. Since the filesystem blocks backing free extents have been removed from the file, at the moment we attempt to write to them again the filesystem has to allocate disk space back, resulting in decreased write performance for the database. This overhead is considered insignificant however compared to unbounded growth of the database file.

4.6 Storing the free space image to disk

In order to avoid data loss and for keeping the database snapshot always consistent, the in-memory data structures must be backed up to disk after each write transaction commits, since write transactions are the ones modifying the database's on disk state.

While storing to disk after each write transaction commit introduces some performance overhead, this overhead is not that significant because:

- Only one of the two trees needs to be stored. Since the two trees are exact opposites a single image can be used to build both of them.
- Only leaf nodes need to be stored. No internal nodes nor the root node.
- The trees will never get too large since they track free extents only.
- Coalescing also reduces tree nodes helping with keeping the tree small.
- Readers stay unaffected, the performance overhead affects only write transactions.

Given that storing only one of the two trees is sufficient the question of which one should be used is raised. The Offset Tree is a good candidate since it is only modified by writers and writers are serialized. This means that nobody is going to modify the tree while we are in the middle of processing a disk commit operation.

The Offset Tree therefore needs to be serialized and stored in a single contiguous extent. For this purpose a few extents of fixed sizes are reserved at certain locations

in the database file. If the image fits in those extents it can be written to any of them as long as it is not the one holding the free space image of the running database snapshot. In case the serialized image is too large to fit in one of the reserved extents, a new extent is allocated and the serialized free space image is written there. A pointer to the extent holding the free space image is stored in the new superblock along with the image size.

4.6.1 Concurrent access to in-memory trees

In the previous section we saw that the Offset Tree is only modified by the writers which are serialized. However, even if we chose to use the Size Tree for storing the free space image there would be no problem either.

Readers do not alter the structure of the Size Tree in any way. Readers may only flip the reclaimable bit flag of extents while a writer is in the middle of a store operation. This is fine though since the information about extent reclaimability is not stored to disk anyway and hence it doesn't affect the operation. There is no point of storing this information given that after a system restart all extents will be considered reclaimable again. Therefore, it is perfectly safe to use the Size Tree for storing the free space image to disk.

This actually also means that no special care needs to be taken for protecting the in-memory B+ trees from data race conditions. No mutex or other lock mechanism is required that would otherwise force writers to block readers and vice versa annulling the concurrent readability property of the database.

4.6.2 Database startup

Upon database startup the free space structures need to be built in memory before the system is ready to operate again. The superblock pointer to the free space image is followed and both trees can be built from the stored nodes. The reclaimable bit is set to 1 for all extents in the Size Tree since there are no read transactions currently and all extents are available for allocation.

The fact that there is no need to scan the whole database image for rebuilding the structures satisfies our fast startup requirement. Additionally, bulk loading can be performed for quickly building B+ trees in memory from only the leaf nodes, making this operation even faster.

4.6.3 Consistency assurance

The write transaction is committed by updating the superblock's priority but this happens only after the free space image has been written to some extent and there is a pointer in the superblock that points to it.

If the system crashes during an allocation operation or at any time before the new superblock is written and its priority is updated, it's like the last transaction never happened. All extents that had been allocated during the last transaction will be considered free and reclaimable after the system comes up again because the last free space image will be used to construct the trees in memory. Any data content changes during the transaction that crashed will be for all purposes invisible. In all cases the database trees and the free space image will be consistent.

4.7 Disk extent header consistency

Extent sizes stored in extent headers on disk are not always valid for free extents. For example, a crash may occur during a coalescing operation after the allocator has updated the size field of the preceding extent but before the transaction committed. That would leave the preceding extent's on-disk size header field with a value larger than its real size.

On the other hand, extent sizes for allocated extents are always valid and this is the reason why they can be used in the deallocation operation. These sizes are always correct because the allocator updates them just before allocating the extent and committing the transaction. If the transaction crashed, the extent wouldn't be considered as allocated. After the transaction has properly committed, the size field is never altered by the allocator before the extent becomes free again.

Not having valid on-disk sizes for all extents has an important consequence. It means that the in-memory free space structures cannot be rebuilt by scanning the whole database image. Consequently, it is safer to store the serialized free space image in more than one places for data redundancy purposes. For increased security a checksum of the image can be stored on the superblock as well.

4.8 Minimized disk accesses

Having the free space data structures reside in memory results in minimal disk accesses. Specifically, only a single disk access is required during an extent allo-

cation for updating the on-disk size field. Similarly, on deallocation of extents a single access occurs in order to read the size field from disk but no accesses for writing data are needed at all. Another disk access is required in order to write the memory state on disk when a write transaction commits. Finally, the coalescing operation can cause more disk accesses but this only happens periodically.

Chapter 5

Conclusions and Future Work

This thesis has thoroughly approached the research topic of persistent dynamic storage allocation for concurrently readable crash resilient database systems. First, we explored the characteristics and the mechanisms of the new key value database. We then conducted a literature survey on memory allocation techniques. Finally, we presented a theoretical design for the extent allocator of the database system satisfying all of the requirements set.

We showed that B+ trees can be used in order to track free space allowing for fast allocation and deallocation of extents in logarithmic time. Maintaining those free space data structures in main memory results in minimal disk accesses.

The allocator takes into account the concurrent readability of the database and its transaction isolation mechanisms distinguishing over liberation and reclamation of extents. We proved that when this API is used properly it ensures that no live storage is ever reclaimed. Furthermore, the allocator allows read transactions to declare their extents as reclaimable in an asynchronous way. Extents reclaimed by the allocator are also deallocated from the filesystem.

Finally, the allocator is resilient to system crashes. The memory state is written to disk whenever a write transaction commits, without overwriting the last free space image. This ensures that there is always a consistent snapshot of the database and the free space map. Additionally, system start is reasonably fast since there is no need to scan the whole database image in order to rebuild the in-memory structures. Only the extent holding the free space image is accessed and the structures are rebuilt quickly thanks to an available B+ tree bulk loading algorithm.

Our future work and research will focus on making extent allocations faster and reducing the overhead introduced by serializing and storing the in-memory structures to disk. We want to determine whether treating small extents specially could increase allocation performance. Specifically, extents of size 4K are the ones allocated most frequently since they are used for storing tree nodes. Maintaining recently freed extents in a linked list instead of directly putting them back in the trees could probably speed up allocations. Another thing to be considered is never coalescing these extents. Furthermore, we want to examine the possibility of using differential updates when storing the free space image on disk in order to make this operation faster and increase the write performance of the database.

Bibliography

[1] Carter Bays. "A Comparison of Next-fit, First-fit, and Best-fit". In: *Communications of the ACM* 20(3) (1977).

[2] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. "Hoard: A Scalable Memory Allocator for Multithreaded Applications". In: *ASPLOS IX Proceedings of the 9th international conference on Architectural support for programming languages and operating systems* (2000).

[3] Howard Chu. "MDB: A Memory-Mapped Database and Backend for OpenLDAP". In: *LDAPCon* (2011).

[4] Yoo Chung and Soo-Mook Moon. "Memory Allocation with Lazy Fits". In: *ISMM '00 Proceedings of the 2nd international symposium on Memory management* (2000).

[5] Webb T. Comfort. "Multiword list items". In: *Communications of the ACM* 7(6) (1964).

[6] Jason Evans. "A Scalable Concurrent malloc(3) Implementation for FreeBSD". In: *BSDCan* (2006).

[7] *fallocate(2) manual. The Linux man-pages project.* URL: `http://man7.org/linux/man-pages/man2/fallocate.2.html`.

[8] Dirk Grunwald and Benjamin Zorn. "CustoMalloc: Efficient Synthesized Memory Allocators". In: *Software - Practice and Experience* 23(8) (1993).

[9] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. "Improving the Cache Locality of Memory Allocation". In: *PLDI '93 Proceedings of the ACM SIGPLAN conference on programming language design and implementation* 28(6) (1993).

[10] David R. Hanson. "Fast Allocation and Deallocation of Memory Based on Object Lifetimes". In: *Software - Practice and Experience* 20(1) (1990).

[11] Thomas E. Hart. "Comparative Performance of Memory Reclamation Strategies for Lock-free and Concurrently-readable Data Structures". University of Toronto, 2005, pp. 7–16.

[12] Yusuf Hasan and Morris Chang. "A study of best-fit memory allocators". In: *Computer Languages, Systems and Structures* 31(1) (2005).

[13] Yusuf Hasan and Morris Chang. "A tunable hybrid memory allocator". In: *Journal of Systems and Software* 79 (2006).

[14] Valtteri Heikkilä. "A Study on Dynamic Memory Allocation Mechanisms for Small Block Sizes in Real-Time Embedded Systems". University of Oulu, 2012.

[15] Arun Iyengar. "Scalability of Dynamic Storage Allocation Algorithms". In: *FRONTIERS '96 Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation* (1996).

[16] Arun Iyengar, Shudong Jin, and Jim Challenger. "Techniques for efficiently allocating persistent storage". In: *Journal of Systems and Software* 68 (2003).

[17] Mark S. Johnstone and Paul R. Wilson. "The Memory Fragmentation Problem: Solved?" In: *ISMM '98 Proceedings of the 1st international symposium on Memory management* (1998).

[18] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Prentice Hall Software Series, 1988, pp. 185–189.

[19] Michael Kerrisk. *The Linux Programming Interface. A Linux and UNIX System Programming Handbook*. 1st ed. No Starch Press, 2010. Chap. 7, pp. 139–152.

[20] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. "Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL". In: *15th USENIX Conference on File and Storage Technologies* (2017).

[21] Kenneth C. Knowlton. "A Fast Storage Allocator". In: *Communications of the ACM* 8(10) (1965).

[22] Donald E. Knuth. *The Art of Computer Programming*. 3rd ed. Vol. 1: Fundamental Algorithms. Addison-Wesley, 1997. Chap. 2.5, pp. 435–456.

[23] Douglas S. Lea. *A Memory Allocator (Dlmalloc)*. URL: `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[24] *Malloc Internals. Glibc wiki*. URL: `https://sourceware.org/glibc/wiki/MallocInternals` (visited on 04/02/2018).

[25] Miguel Masmano, Ismael Ripoll, Patricia Balbastre, and Alfons Crespo. "A Constant-Time Dynamic Storage Allocator for Real-Time Systems". In: *Real-Time Systems* 40(2) (2008).

[26] *mremap(2) manual. The Linux man-pages project*. URL: `http://man7.org/linux/man-pages/man2/mremap.2.html`.

[27] Sathish K. Palaniappan and Pramod B. Nagaraja. *Efficient data transfer through zero copy. Zero copy, zero overhead*. IBM developerWorks website. Sept. 2, 2008. URL: `https://www.ibm.com/developerworks/library/j-zerocopy/index.html`.

[28] Thomas A. Standish. *Data Structure Techniques*. Addison-Wesley, 1980.

[29] C. J. Stephenson. "New methods for dynamic storage allocation (Fast Fits)". In: *ACM SIGOPS Operating Systems Review* 17(5) (1983).

[30] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. "Dynamic storage allocation: A survey and critical review". In: *International Workshop on Memory Management* (1995).

[31] *XFS Filesystem Structure. Documentation of the XFS filesystem on-disk structures*. 2006. URL: `http://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure/tmp/en-US/html/index.html`.

[32] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. "Dynamic Memory Optimization using Pool Allocation and Prefetching". In: *ACM SIGARCH Computer Architecture News* 33(5) (2005).