Hasso Plattner Institute
Chair for Data Engineering Systems

Master Thesis

R-tree Data Placement on Persistent Memory

Datenverteilung auf Persistentem Speicher für R-Bäume

Nils Hendrik Thamm

Matriculation Number:

Supervisor
Prof. Dr. Tilmann Rabl

Second Supervisor
Prof. Dr. Felix Naumann

Advisor
Lawrence Benson

Submitted: 24.06.2021

Abstract

Byte-Addressable Persistent Memory (PMem) is a new memory technology that offers persistent storage with an access latency close to DRAM. Because PMem shows different performance characteristics than DRAM, it cannot replace DRAM for all applications. For index structures in database systems, the combination of persistent storage and low access latency is well suited. During runtime, the index needs to process queries as fast as possible. After a restart, the index needs to be rebuilt from the complete dataset if it is not persistently stored.

Existing persistent index structures are designed for storage mediums with block-based access. The byte-addressability of PMem requires a redesign of all these structures. All existing proposals for PMem aware persistent index structures either do not use DRAM at all or use it without any consumption limit. While DRAM has a better access performance, it is also more costly than PMem and available in smaller sizes. Therefore, we propose persistent adaptive R-Tree (PeaR-Tree), a PMem aware R-tree index supporting *adaptive persistence*. *Adaptive persistence* is a novel concept that allows setting a DRAM consumption limit. PeaR-Tree dynamically places parts of the structure in DRAM respecting the limit. This thesis presents the implementation of PeaR-Tree and how it realizes *adaptive persistence*. Our evaluations show that with more DRAM consumption, the runtime performance increases proportionally. In the same proportion, the recovery time increases. PeaR-Tree allows controlling this trade-off on a fine granular level. In addition, it outperforms other state-of-the-art persistent R-tree structures, both designed for SSDs and PMem.

Zusammenfassung

Byte-adressierbarer persistenter Speicher (PMem) ist eine neue Speichertechnologie, die annähernd die gleichen Zugriffszeiten wie DRAM hat. Auf Grund einer unterschiedlichen Performanceausprägung kann PMem allerdings nicht als direkter Ersatz für DRAM genutzt werden. Ein sinnvolles Einsatzgebiet von PMem sind Indexstrukturen in Datenbanken. Während der Laufzeit muss ein Index Anfragen so schnell wie möglich beantworten. Nach einem Systemneustart muss die gesamte Struktur von Grund auf neu erstellt werden, wenn sie nicht vorher persistiert wurde.

Aktuelle persistente Indexstrukturen sind für block-basierte Speichermedien entwickelt worden. Die Byte-Adressierbarkeit von PMem erfordert, dass diese Indexstrukturen angepasst werden müssen. Alle bisher veröffentlichten PMem-Indexstrukturen setzen DRAM entweder gar nicht oder ohne ein spezifisches Limit ein. Einerseits ist es von Vorteil, die bessere Zugriffsperformance von DRAM zu nutzen. Andererseits ist DRAM teurer und nur in kleineren Speichergrößen verfügbar. In dieser Arbeit präsentieren wir den adaptiv-persistenten R-Baum (PeaR-Tree). In unseren Messungen zeigt sich, dass sich mit einem größeren DRAM-Limit die Performance proportional erhöht. Gleichermaßen erhöht sich die Wiederherstellungszeit nach einem Systemneustart. PeaR-Tree ermöglicht es, diese Wechselwirkung fein granular zu steuern. Darüber hinaus übertrifft es sowohl für SSDs optimierte als auch für PMem optimierte herkömmliche persistente R-Baumstrukturen.

Contents

1 Introduction

Access latency for persistent storage has decreased drastically. While the cache and DRAM are still the fastest ways to access and store data, SSDs have done a great deal to close the gap between volatile memory and non-volatile storage. Already the first SSD, in 2007, achieved access latency of 0.5 ms over the best disks with 7.7 ms at that time [7]. Still, DRAM read latency is around 80 ns, making DRAM orders of magnitudes faster [51]. Today the best commercially available SSDs achieve a read latency of 12-45 μs, which is still three orders of magnitude slower [52]. It shows, there is still a significant gap between DRAM and SSDs. However, a new technology emerged, which can be described as non-volatile memory. Combining the speed of DRAM with persistence closes exactly the gap between DRAM and SSD.

PMem offers byte addressability while being persistent even in the case of power failure. In research PMem has already been a topic for several years, even though Intel only released the first commercially available PMem in 2019 [1]. With the *Optane DC Persistent Memory* DIMM (Optane DIMM), Intel proves that PMem indeed closes the gap between SSDs an DRAM. Its access latency is between two to three times slower than DRAM. Also, it only achieves half of the DRAM bandwidth [5, 16, 53, 54]. Still, PMem has some advantages over DRAM. First, being persistent and byte-addressable at the same time allows new designs of persistent data structures. The cost of persistence is drastically reduced since no data replication is necessary anymore to achieve persistence. Second, with persistence, the start-up time of applications can be lowered to a fraction of the start-up time from a pure in-memory application. For example, SAP offers the option to use PMem for its in-memory database HANA. In their setup, the PMem does not replace the main memory completely but for about 90% [49]. Considering that a HANA instance can use up to 2 TB of main memory, the saved time is in the range of minutes [10]. Third, Optane DIMMs are cheaper per GB and offer more space per DIMM than conventional DRAM DIMMs. The release price for an Optane DIMM with a capacity of 512 GB was about \$6700 (~\$23/GB) while the largest available DDR4 DIMM only offers 256 GB for around \$6000 (~\$35/GB), which is factor 1.5 more expensive and factor 2 smaller [23].

One of the major challenges for PMem compatible applications is failure-consistency. PMem applications need to ensure that a write does not produce a corrupted data state if the application crashes directly afterward or even in between. Therefore, PMem cannot just be used as a replacement for DRAM but has to be integrated into the storage hierarchy of an application. Since PMem is byte-addressable, new methods for failure-consistency and recovery have been developed in favor of existing mechanisms for disk-based persistence to keep the overhead as small as possible [13, 36, 37, 40, 55]. On top of this overhead, current research suggests that the performance characteristics differ from DRAM. The latency difference for random access versus sequential access is relatively higher. Additionally, the bandwidth highly depends on the access size and does not scale well with multiple write-heavy workloads. Nevertheless, the benefit of using PMem depends on the use case [16, 53].

As the SAP HANA example already demonstrates, big database systems can leverage the advantages of PMem well. Consequently, much research focuses on the use case of PMem in the context of database systems. One critical part of database systems are their indexes. On the one hand, they need to be as efficient as possible to speed up access to the underlying data. On the other hand, they are costly to construct and need to touch each entry at least once during construction. This forces database system developers to choose between fast but volatile in-memory indexes and slow but persistent on-disk indexes. Therefore, different works proposed PMem index structures. They offer instant or near to instant recovery and the performance is close to in-memory indexes [13, 36, 37].

However, all of these index structures are only suitable for one-dimensional data spaces. Multi-dimensional data spaces pose different requirements towards an index in order to be efficient. For one thing, they can contain objects of non-zero size, which also can have an overlap between each other. For another, multi-dimensional predicates include spatial relationships between objects, such as distance. Both properties cannot be handled well by one-dimensional indexes.

Spatial databases are one use case of multi-dimensional indexes. They describe all database systems, which need to process spatial relations between their objects. A common example would be a geospatial database storing, e.g., the historical rides of a taxi provider. Apart from information like the driver's name or ride's fare, the start- and endpoint would be saved as well. If a user would now like to query the five rides, which started the closest to a given point, a multi-dimensional index is needed not to find the rides efficiently. A one-dimensional index could not limit the search space well, given the predicate of five closest entities to a point.

The R-tree is a prominent multi-dimensional index structure. It is used by many database systems (e.g. boost [18], Oracle [34], IBM [26], PostgreSQL [19], and MySQL [44]). Compared to other multi-dimensional index structures, it offers excellent flexibility. An R-tree can equally efficient index data of any dimensional size and objects of zero and non-zero size. Many variations have been developed, but they all rely on the same principle. The core principle of an R-tree is to divide the data space into rectangular subspaces recursively. Each recursion forms a new node of the tree so that a branch in a node only needs to save the bounding box of the child node. Subspaces are allowed to overlap, making the division and extension of subspaces less complex.

On the other hand, the overlap requires loading into memory and reading all branches of a node during a query operation. Therefore, node placement in the storage hierarchy is even more crucial for the R-tree than for other tree index structures. Apart from variations of the R-Tree, which improve the insert or split algorithms, proposals have been made to adapt the R-tree to flash-memory-based systems or SSD-based systems [30, 38]. Only Failure-Atomic Byte-Addressable R-Tree (FPR-Tree) was proposed as an R-tree designed for PMem to the best of our knowledge. However, FPR-Tree exclusively runs in PMem. Unlike other persistent R-Trees, it does not keep a working set in main memory. FP-Tree showed that it is beneficial to only store parts of an index structure in main memory [45]. It introduced the concept of selective persistence, where only leaf nodes are persisted in PMem and inner nodes lie in DRAM. With selective persistence, the index has a lower access latency because parts of it are located in DRAM. Still, it can be recovered entirely by rebuilding the inner nodes based on the persistent leaf node structure.

In this thesis, we present PeaR-Tree. PeaR-Tree offers *adaptive persistence* so that all entries in the leaves are persisted, but inner nodes may not be. Thereby, the degree of persistence is not statically limited to the leaf level. It can be chosen dynamically. This allows the PeaR-Tree to adapt to the available amount of DRAM and leveraging the improved access times as much as possible. The challenges arising from that setup are manifold. First, operations on nodes need to be failure-consistent. Analogous to to the work of Cho et al. and other persistent index structures, this includes redesigning the operations itself as well as the data layout of the nodes [13, 36, 37]. Second, operations on persistent nodes need to reflect the performance characteristics of PMem. Third, not only operations on nodes but operations on the tree structure need to be failure-consistent. Especially for the parts of the tree where persistent and volatile nodes are connected, this poses a new challenge. Finally, the recovery process needs to handle only partially persisted tree structures and recover them as efficiently as possible. Sections 4 and 5 cover how we address these challenges. Overall, with our findings we provide answers to the following three key questions:

- How can an adaptively persisted index structure be implemented?
- How can an adaptively persisted index structure be recovered efficiently?
- What is the influence of *adaptive persistence* on the run-time performance of an index structure?

1.1 Motivating Example

As a practical example of a large-scale spatial data use case, we use the NYC Yellow Cab historical ride data throughout this thesis. The NYC Taxi & Limousine Commission provides data for all rides conducted since 2009 [42]. Per month there are on average about 10 million rides. Over 12 years, this results in over 1.5 billion rides. For each ride, a start- and endpoint is provided, along with other information.

We assume we want to analyze the best course of action for a taxi driver after he dropped off a passenger. The question is, if it is better to stay in the area waiting for a new passenger or drive back to the taxi depot to wait there. In order to understand the problem better, we want to know where and when new passengers were picked up in close proximity to a drop-off location. It is not important which taxi picked them up, only that it happened after the drop-off of a given ride. An example query would be: Return the ten closest rides to the drop-off point of ride 42, which have a pick-up time larger than the drop-off time of ride 42. In order to answer that query as quickly as possible, the database needs a multi-dimensional index. In contrast to a one-dimensional index, it preserves the spatial locality.

Given the NYC Yellow Cab dataset example, the points are encoded in single precision floats, i.e. 4 B. Therefore, an index on the start- and endpoint information would occupy at least 45 GiB. On the one hand, an in-memory index would offer a low access latency but require a rebuild after every start-up. Thus, it would increase the start-up time. On the other hand, an on-disk index would not influence the start-up time since it is stored on a persistent medium. But it would have a high access latency. A database administrator would have to trade-off access latency for start-up time or the other way around. Considering the size of the data set, it would either be index access latency in the millisecond range or multiple minutes of start-up time alone to build the index. On top of that, an in-memory index would already occupy a large portion of costly DRAM.

With the availability of PMem, the trade-off does not have to be made. PMem is cheaper than DRAM and only has at most two times worse access performance. An index ran entirely in PMem could be recovered instantly with two times longer access latency than an in-memory version. However, in combination with DRAM, the trade-off on the latency can be reduced further by storing parts in DRAM, not occupying the full size of the tree. Especially in in-memory database systems, where DRAM capacity is a critical resource for multiple components, a partial persisted index can offer flexibility at maximal performance.

1.2 Contribution

We propose PeaR-Tree to offer a more flexible fit for the gap between volatile in-memory indexes and persistent on-disk indexes. With the use of PMem, PeaR-Tree is a persistent index structure, which can be recovered orders of magnitude faster than in-memory trees. Combining the persistent part dynamically with a volatile part in-memory, PeaR-Tree offers *adaptive persistence*. Adaptive persistence guarantees the persistence of data values in the leaf nodes. Other parts of the tree may only be held in DRAM. While they are not persisted, they have a low access latency. In contrast to selective persistence, the degree of persisted nodes can be configured dynamically. With this novel approach, we make the following contributions:

1. We present Persistent adaptive R-Tree (PeaR-Tree), a concurrent, dynamic, persistent R-Tree, ensuring failure consistency on PMem.
2. We derive Guidelines to implement adaptive persistent index structures on PMem.
3. We evaluate PeaR-Tree and compare it against state-of-the-art in-memory and persistent R-Trees. With the results, we show that it outperforms existing persistent versions by up to orders of magnitude and comes close to pure in-memory versions.

1.3 Scope and Delimitation

We investigate *adaptive persistence* for the multi-dimensional index R-Tree. Therefore, we implement persistent adaptive R-Tree, which satisfies all requirements of a two-dimensional R-Tree. Its algorithms for insertion and splitting are based on state-of-the-art research. We did not further investigate modifications to these existing functions. In the context of these systems, which are necessary to use PMem, we also support concurrent operations. However, we refrain from supporting higher dimensions. They are independent of the concept of *adaptive persistence* and only require optimizations regardless of the data placement. The offered interface only supports the benchmarked query predicates, and it only supports failure resistance against PMem errors, but not against corrupt user input or other failures.

1.4 Thesis Outline

Section 2

In Section 2, we introduce all relevant concepts and explain the foundation for this thesis. First, we describe the concept of PMem and its technical details. Second, in the context of possible use cases, we introduce spatial databases and their differences from regular databases.

Section 3

In Section 3, we describe the R-Tree. We explain each operation on a conceptual level, which is the same for most of all R-tree variations. Then, we give an overview of the related work for R-Trees, including a detailed presentation of the FPR-Tree.

Section 4

Section 4 presents the fully persistent configuration of PeaR-Tree. First, we describe the architecture and in detail the persistent node structure. Based on that configuration, we prove the failure consistency of all node operations. Second, we develop the recovery process.

Section 5

Following the presentation of the fully persistent PeaR-Tree, we illustrate the design of an adaptive persistent index structure on the example of the PeaR-Tree in Section 5. This includes the different concepts of *adaptive persistence*, the procedures to guarantee an adaptive persistent tree structure at run-time and the recovery of adaptive persistent index structures.

Section 6

In Section 6 we present our evaluation results. On the one hand, the evaluations include a comparison against state-of-the-art in-memory and persistent R-tree versions. On the other hand, the evaluations form an extensive analysis of the trade-offs coming with *adaptive persistence*. They include micro-benchmarks of different setups, where the possible configurations of the PeaR-Tree are compared against each other.

Section 7

Finally, in Section 7, we summarize our findings for the work on adaptive persistent tree index structures. Additionally, we give an outlook on future research and applications.

2 Background

This chapter gives an overview of the essential concepts and technical details connected to this thesis. First, we describe the technical foundation and differences from PMem to existing storage technologies in Section 2.1. Second, we give an overview of the resulting challenges for users and in which cases PMem is of use. One of the applicable fields are database systems. In specific, we look at spatial database systems, a specialized group of database systems. Section 2.2 differentiates spatial database systems from database systems in general. Going into more detail, we explain in Section 2.2.1 how multi-dimensional indexes are different from one-dimensional indexes.

2.1 Byte-Addressable Persistent Memory

PMem is a new class of storage technologies bridging the gap between volatile memory devices, like DRAM, and persistent storage, like HDD and SSD. While the concept is already known in research for years, Intel only recently released the first publicly available device, the Optane DIMM. They are based on 3D XPoint technology, which Intel also uses in some of their SSDs. Compared to DRAM DIMMs, 3D XPoint based devices are built denser. Therefore, they offer higher capacities per DIMM. Moreover, they are available at a lower price per GB. Since the Optane DIMM is the only available PMem device at the time of writing, we only focus on its characteristics when speaking about PMem and use the two terms interchangeably.

Similar to a DRAM DIMM, an Optane DIMM is connected to the CPU through a memory channel and the integrated memory controller (iMC). Depending on the CPU architecture, a CPU socket can have multiple iMCs. Whereas all cores of a socket can access all iMCs, an Optane DIMM is only connected to one iMC. Figure 1 shows an exemplary setup of the *Cascade Lake* CPU Architecture, where all channels are fully utilized. The *Cascade Lake* CPUs allow one Optane DIMM per memory channel in addition to the DRAM DIMM. Since its iMCs each support three memory channels, a total of six Optane DIMMs can be connected.

When multiple Optane DIMMs are connected, they can be set to interleaved access. With interleaved access, the memory region is separated into blocks of 4 KB. Each block resides on a different Optane DIMM than the block with an adjacent address range. Hence, data objects larger than 4 KB can be read in parallel because each 4 KB block of the objects is located on a different Optane DIMM. E.g., a 24 KB large object would be stored across all six Optane DIMMs.

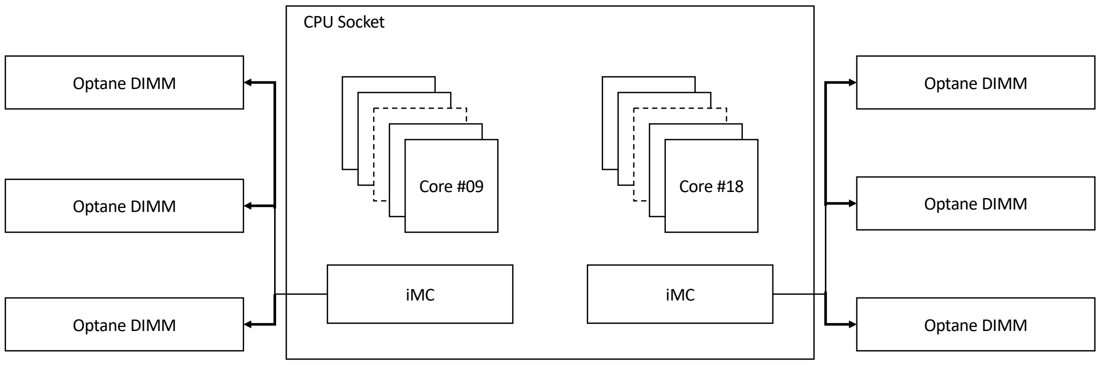


Figure 1: Examplary *Cascade Lake* setup with six Optane DIMMs.

PMem can either be run in *App Direct Mode* or *Memory Mode*. In *Memory Mode* the DRAM functions as an additional cache and the PMem functions as main memory. This mode allows applications to be run PMem without further modifications since it appears as regular DRAM to the application. As PMem is available in larger DIMM sizes than DRAM, it is an easy way to extend the main-memory capacity. On the other side, data is not persisted.

The *App Direct Mode* allows the user to control and access the PMem directly. To the CPU, the PMem appears as a separate memory device and can therefore be used as such. In this mode, the PMem functions as a persistent device, also in case of a shutdown or power failure. To leverage the benefit of byte-addressability, PMem is commonly accessed through memory-mapped files. Through functions like *mmap(2)*, users can map the content of a file into the virtual address space of their application. In the background, the operating system flushes changes made in memory to the file location. If this file is located on a PMem device and the device is mounted in *devdax*-mode (*Device-DAX*-mode), the operating system does not need to do any regular file system management like page buffering. Instead, an evicted dirty cache line is directly flushed through the iMC and read accesses require no buffering overhead. However, as a result, a change to a memory-mapped file in PMem is only persisted if the change-containing cache line gets evicted. Otherwise, the change is lost in case of a failure, because only the iMC is part of the asynchronous DRAM refresh domain, which guarantees persistence even in case of a power failure. Since only the *App Direct Mode* offers persistence, we assume PMem is run *App Direct Mode* in the remainder of this thesis.

Consequently, developers have to treat data modifications in PMem differently than data modifications in DRAM. Not only could changes be lost, but also corrupted data can be persisted if a crash happens within a write of a data object. If a change should be persisted immediately, the developer can issue a cache line flush containing the modified object. Intel offers the cache line write back (*clwb*) function for that use case. It flushes the cache line but does not evict or invalidate it. Subsequent accesses to the same address therefore do not produce a cache miss [28]. Since the CPU can reorder memory operations as long as the logical integrity is not compromised, developers need to use fencing instructions to ensure writes to PMem happen in the designated order. Fencing instructions, such as *sfence*, introduce memory barriers, over which the CPU cannot reorder instructions [29].

Although a cache-line is flushed all at once, the CPU only guarantees word-size atomicity, usually 8-byte. Different approaches have been developed to write objects larger than 8-byte recoverable. The NV-Tree, one of the first persistent B-Tree structures, organizes its node like an insert-only log [55]. For each modification, a new entry is added. A flag indicates if an existing entry is now deleted or if a new entry was added. However, this requires always reading the entire node whenever its entries are accessed. CCEH, a consistent hash index, mitigated this problem by using copy-on-write mechanisms for bucket splits and commit marks on each entry during insert [40]. If a crash happens, entries without a valid commit mark are disregarded and uncompleted splits can be reconstructed since no data was overwritten. The redesign of CCEH, Dash, reduces the additional writes by combining a bitmap and state indicating variable [37]. Entries are only valid if their corresponding bit is set. Depending on the state of the node, operations know if they need to consider other buckets. Because the bitmap and the state indicating variable are combined in an 8-byte word, they can be updated atomically. Therefore, the node changes atomically between states and operations done between the state transitions only become valid afterward.

Generally speaking, PMem performance is closer to DRAM performance than to SSD performance. However, its behavior is more nuanced for different access patterns than for DRAM access. Research shows that the maximum bandwidth increases drastically up to an access size of 4 KB and plateaus for larger access sizes [16, 53, 54]. The actual maximum bandwidth depends on the setup. Research suggests that with six Optane DIMMs, a maximum bandwidth close to 40 GB/s can be achieved. On the same systems, DRAM performs more than twice as good [5, 16, 53, 54].

The inconsistent behavior can again be observed when we compare the latency of random and sequential accesses. For sequential write access, DRAM and PMem perform about the same. Sequential read access shows a higher latency up to a factor of two. On the other hand, random reads on PMem have a three times higher latency than random reads on DRAM. In some cases, random writes on PMem even show a five times higher latency than random writes on DRAM [5, 54]. The difference comes from a deviating access granularity. Although PMem is byte-addressable, like DRAM, the granularity of physical accesses is different from physical accesses to DRAM. While DRAM can be accessed on cache line-granularity, in most cases 64 B, the physical access to PMem happens in 256 B blocks. The translation between the different granularities happens in the PMem DIMM itself. Each iMC maintains queues for pending read and write requests, the read and write pending queues. This allows the iMC to bundle access smaller than 256 B to the same block together and only access the block once. If no accesses can be bundled, the whole block is loaded. Therefore, the consequence for random accesses is a high amplification.

2.1.1 Related Work

Apart from the general research for PMem, much research focused on developing PMem aware index structures [2, 11, 12, 13, 36, 37, 40, 45, 55]. Big database systems are suitable for the use of PMem because of their large main memory consumption. Indexes benefit from the persistence and still good performance since they are costly to construct and need to offer low access times.

FP-Tree is a tree index structure with selective persistence [13]. In a selective persistent index structure, only the parts that are essential to recover the structure are persisted. For a tree index structure, these are the nodes storing data values. In most cases, only the leaf nodes are persisted with selective persistent tree index structures. The inner nodes, which can be reconstructed based on the leaf nodes, lie in DRAM [45]. This allows leveraging as much DRAM performance as possible while still being persistent. The leaf nodes are stored unsorted to avoid rewriting the leaf entries on every insert. While an unsorted node needs to be scanned entirely for every query, FP-Tree uses fingerprints to mitigate unnecessary reads of keys. Fingerprints are a hash with a size of 1 B. Before a leaf is accessed, the fingerprint is probed against the predicate. Due to their small size, it is less expensive to load the fingerprint into main memory than the actual key. Further does the FP-Tree proposes amortized allocation. Since allocations on PMem are more expensive than on DRAM, they allocate memory for the nodes in larger chunks. Deallocated regions of a chunk are placed in an array. New nodes are initialized with a pointer from the array. If the array is empty, a new chunk gets allocated.

Dash is an extendible hash index optimized for PMem [37]. Opposed to earlier hash indexes for PMem it is tested on a real PMem device. Hash indexes build on simulated PMem failed to reduce the number of reads to PMem and only reduced the number of writes. Therefore, Dash uses fingerprinting. To postpone segment splits, which require many write operations, they employ a bucket load mechanism at insert as well as overflow buckets. Dash does not use selective persistence but is fully stored in PMem. Due to a state concept of the nodes, an instance can be recovered in constant time. Operations that are accessing a node during runtime check if the node is still in an inconsistent state. An inconsistent state occurs if an operation did not finish modifying the node before the last restart.

uTree identifies the tail latency as one of the major bottlenecks of earlier work [12]. To improve the access tail latency, uTree uses a shadowed linked list of all leaf nodes in addition to the leaf level in the tree. The linked list can be resorted with fewer write operations than the array-like tree node. uTree adopts selective persistence with storing the linked list and the leaf nodes in PMem and all other nodes in DRAM. However, because of the sorted shadow leaf list, uTree does not apply fingerprinting.

Opposed to other persistent tree index structures, LB+-Tree optimizes the structure and operations for cache line size access [36]. Their observation is that changes to a cache line have the same persistence costs no matter their size. Therefore, they structure the node layout in a way that changes touch as few cache lines as possible. They still use fingerprinting and selective persistence.

Nearly all PMem index structure research focuses on one-dimensional index structures like hash tables and B-Trees. To the best of our knowledge, only one proposal for a multi-dimensional PMem index structure exists, Failure-Atomic Byte-Addressable R-Tree [13]. We discuss the FPR-Tree in the context of R-trees in Section 3.5.1.

2.2 Spatial Database Systems

Database systems, in general, are a good use case for PMem. For the best performance, they need to hold as much data in main memory as possible. Using PMem as main memory still offers good access performance and allows to omit flushes to persistent storage layers. While this is true for all database systems, we focus on a particular group of database systems, the spatial database systems. Spatial databases have the same capabilities as non-spatial databases, only that they are optimized to store and query spatial data. To demonstrate the relevance of these optimizations, we consider again the NYC Yellow Cab taxi data set [42].

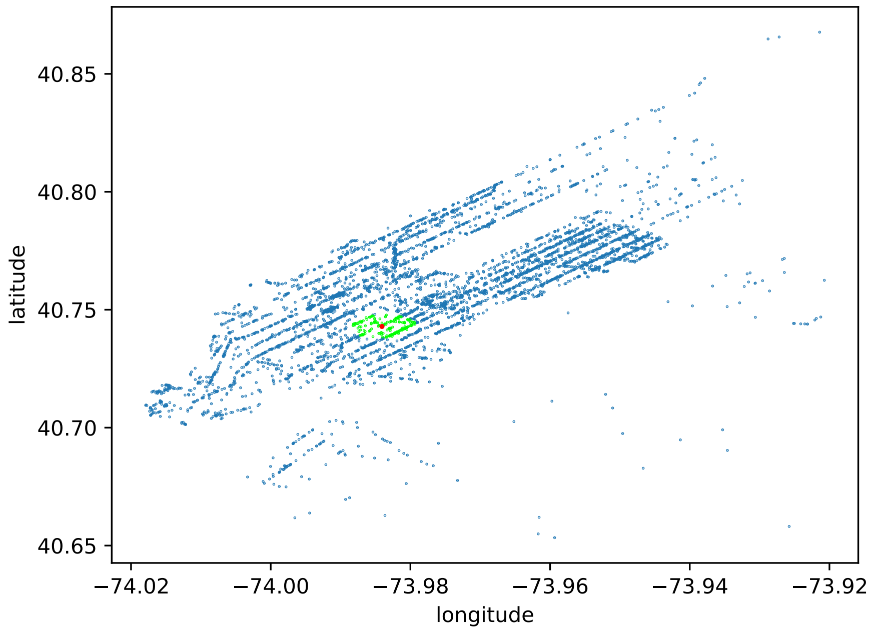


Figure 2: NYC Yellow Cab Taxi data set plot of 5000 start points.

Figure 2 shows a 2D-grid visualization of 5000 startpoints from the taxi data set. If we consider the query: Return the ten closest rides to the drop-off point marked in red, the colors indicate the points a database would calculate the distance for with and without a multi-dimensional index. Without the multi-dimensional index, a database would need to calculate the distance to the point in red from all points to identify the ten closest. However, with a multi-dimensional index, the database would need to calculate far fewer distances. Assuming the multi-dimensional index preserves the spatial locality, the database can iteratively extend the search space. At a certain point, index groups are too far away to contain points that could be closer than the already found ones. The points colored in green show an approximation of how few points a database would calculate distances for.

While these optimizations are not visible for the user, spatial databases also offer users ways to describe objects of different shapes and sizes in space [21]. Most spatial database systems or spatial extensions to existing systems do this by supporting common geometrical shapes as an own data type (e.g., IBM Informix Spatial DataBlade Module [25], Oracle spatial database [43], PostGIS for Postgres [46], and SQLAnywhere within SAP HANA [20]). Further users need to be able to describe spatial relationships in queries [21]. Although the concept of spatial databases is not limited to relational databases, most spatial databases are organized as relational databases. On top of the standardized SQL operations, they offer operations and predicates specific to their spatial data types.

While the support of spatial query functions is a question of translating the spatial predicates into relational logic, the efficient execution of these queries requires more additional concepts. Databases dealing with non-spatial data rely on indexes in order to reduce lookup times for given predicates. Indexes achieve the speed-up by structuring the data space, so only very few unnecessary lookups are necessary for a predicate. In a one-dimensional data space, a total order can easily be achieved, allowing the index to structure the data efficiently. In a multi-dimensional data space, the same functions cannot achieve a total order, preventing an efficient structuring. The dimensions could be indexed individually and organized as a multi-layered index. While it would allow more efficient querying than without an index, the causal dependency of locality between the dimension is not leveraged. Therefore, spatial databases use multi-dimensional index structures.

2.2.1 Multi-Dimensional Indexes

Multi-dimensional indexes structure a dataset with a multi-dimensional data space efficiently so that lookups for specific entries need to touch as few unnecessary entries as possible, like one-dimensional indexes. Most one-dimensional indexes use one of two properties of their data space to reduce the number of unnecessarily touched entries: Either they can be put in total order, or a hashing function can be applied. These properties cannot be leveraged as well for multi-dimensional data space.

A hashing function is a function returning a hash for any input value of a given data space in constant time complexity. If two values are equal, their hash is also equal. In an index, a hash function can be used to partition the data space into smaller groups. For a given predicate, only the group that matches the predicates' hash needs to be searched. While it is theoretically possible to hash a multi-dimensional object, the benefit of a multi-dimensional hash index is considerably smaller than for one-dimensional hash indexes. The value range of the data space is exponentially higher in the number of dimensions. Thus it is impractical to partition it into smaller groups.

Another way one-dimensional indexes partition the data space into smaller groups is by leveraging their implicit total order if the keys are natural numbers. Dynamically set ranges define the groups. Objects in a multi-dimensional space have no implicit total order. Only with a defined ordering function, like space-filling curves, a total order can be applied. An early example of such an approach is the Morton space-filling curve. The curve is defined in a way that it covers the whole range of a two-dimensional data space. By calculating the position of a key on the curve, all keys can be put in a total order. With the total order at hand, any known one-dimensional index can be used. Because the curve preserves the spatial locality of the objects in its order, the index can also be used for range queries [39].

However, the use case of multi-dimensional indexes goes beyond simple point and range queries. Instead of a fixed range, multi-dimensional key spaces can also be queried for the closest objects to a given point, so-called associative queries [6]. While an index built based on a space-filling curve can limit the search region compared to the whole data space, other structures were developed to better help these kinds of queries. Instead of applying a total order to use a one-dimensional index, they define a new index structure. The Kd-Tree and quadtree both recursively divide the data space into subspaces on each level [6, 17]. The only difference is the outgoing degree of a node, two for a Kd-Tree and at most four for a quadtree, and how the nodes' points are divided during a split. Because neighboring nodes also cover adjacent data spaces, associative queries can be answered by only visiting very few nodes of the index.

When it comes to multi-dimensional objects of non-zero size, indexes working with space-filling curves and indexes optimized for associative queries both fall short. If a data space contains spatial objects, it has multiple implications for an index covering that space. First, if not enforced by the application, spatial objects can overlap. Second, spatial indexes must allow queries for geometrical relationships, such as contains, covers, or overlaps. Space-filling curves can theoretically represent spatial objects if they treat them as multiple points, but the above operations are complex to implement. A quadtree, for example, can support spatial objects by storing references to objects in multiple nodes. However, this contradicts the purpose of indexes, to structure the data space such that as few as possible entries need to be touched for a query. Out of these short-comings, the R-tree evolved. Its structure represents spatial locality as well as spatial dimensionality [22].

3 R-tree

The R-tree is a multi-dimensional index supporting spatial range queries efficiently [22]. Because of its flexible structure and fit for spatial datasets of all dimensions, while still showing comparable performance, it found great acceptance. Many major database systems implemented variations of it (e.g., boost [18], Oracle [34], IBM [26], PostgreSQL [19], and MySQL [44]).

In this chapter, we present the concepts and operations of R-trees as initially proposed by Guttman [22]. While many variations exist, the core principles are the same. First, we explain the basic structural properties in Section 3.1. Second, we demonstrate how typically index operations work for R-tree in Section 3.2 and following. Finally, we present a selection of adaptions to R-tree and how they differ in Section 3.5. For simplicity, we only consider a two-dimensional data space in the remainder of this thesis. All operations and principles work on higher dimensions as well.

3.1 Structural Properties

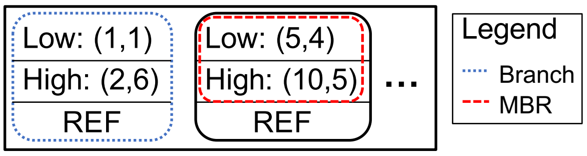


Figure 3: Example R-tree node structure.

An R-tree consists of nodes that have links to each other, forming a tree structure. As shown in Figure 3, each node contains branches. Except for the root node, each node always has at most M branches and at least m branches. M and m are parameters that can be chosen freely for an instance of the R-tree, whereas $2 \leq m \leq M/2$. In the following, we use M to denote the maximum number of branches for a node and m to denote the minimal number of branches for a node. For inner nodes, the branches represent the children of a node. In the last level, the leaf level, the branches hold the references to the data objects. Besides a pointer, a branch holds the information for the Minimal Bounding Rectangle (MBR) of its child. A MBR encloses all objects contained in the subtree under the respective branch. It is the smallest possible rectangle that spatially contains all MBRs of the child node. Since the MBR are rectangles, their shape can be encoded as a high-point and a low-point. While multiple MBRs of the same level are allowed to overlap, it is beneficial to keep the overlap as small as possible. Because of the allowed overlap, a search query always needs to evaluate for all branches in a node if the underlying subtree may contain a match. If multiple MBRs overlap and the overlap falls within the search space, the search needs to include all child nodes of the overlapping branches.

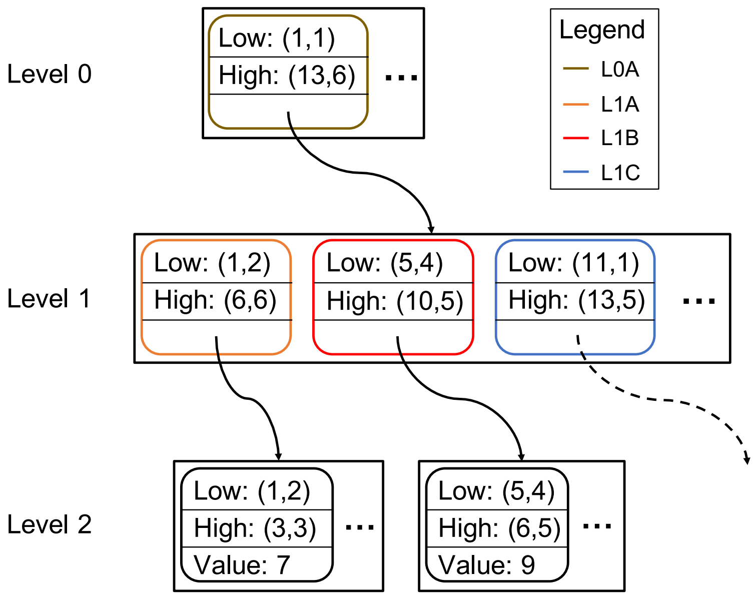


Figure 4: Part of an example R-tree.

Figure 4 shows a part of an exemplary two-dimensional R-tree and Figure 5 shows the corresponding visualization in a 2D-grid. Figure 5 shows how the MBR, *L0A*, of the root branch (compare Figure 4) encloses all sub-tree MBRs and the respective data objects. Although it might look evident from the visualization, an overlap of the MBRs on level 1 is unnecessary. It is not uncommon that an R-tree structure has these artifacts. They occur since the structure of the tree not only depends on the shape of data objects but also on their insertion order, as we explain in Section 3.2. Another artifact of R-trees is that MBRs can cover areas with no values in the underlying data. From Figure 5 it is apparent that this is inevitable when the bounding boxes are rectangles and need to cover all sub-elements. Otherwise *L0A* could not cover *L1A* and *L1C* at the same time.

3.2 Insert

An insert operation consists of two parts. First, the subtree has to be chosen on each level recursively. Second, the object has to be inserted in the leaf and

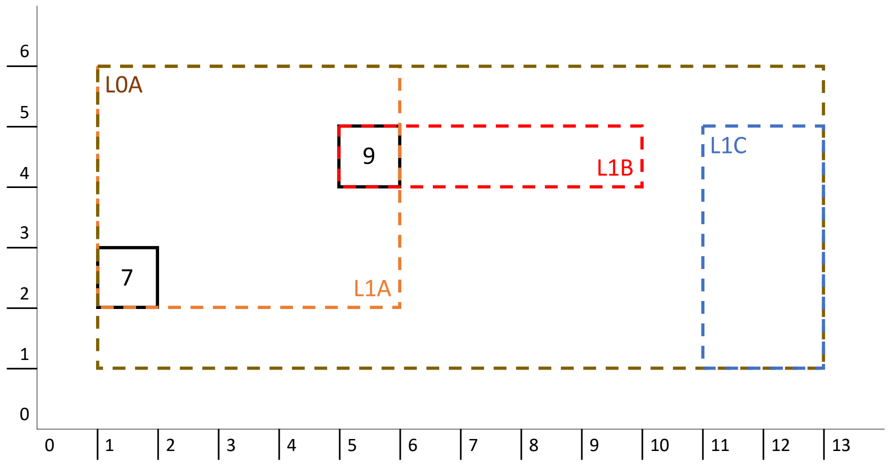


Figure 5: Visualization of the example R-tree from Figure 4 in a 2D-grid.

the tree needs to be adjusted. Adjusting the tree includes updating all MBRs along the chosen path. It includes a possible split of the leaf node and subsequent nodes in higher levels. A split is necessary as soon as an insert happens to a node containing the maximum number of branches. First, we describe how an insert happens without a split before we explain the insert triggering a split.

Figure 6 shows the steps to insert a new object without causing a split in the example R-tree from Figure 5. The routine adds an object with the rectangle $((9,3);(10,4))$ and value 3. Starting at the root node, the routine recursively selects a subtree to insert the new object (①). Different metrics are used to determine the best subtree. The original R-tree chooses the subtree whose MBR needs the least area enlargement to contain the new data object. Experiments show that this algorithm offers a good balance between the cost to find a good candidate for later query performance and initial insert performance. Because the branches in a node are not sorted in any way, the area enlargement needs to be calculated for all MBRs.

From Figure 5 we see that the new object should go into the node with the MBR *L1B*. After a leaf node was found, the value and dimensions are added in the second step (②). The third and final step is to update the tree structure accordingly. Starting from the leaf node, the routine checks for each level if the respective MBR needs to be updated. On level one, the low point y value of the MBR *L1B* needs to be set from 4 to 3 to fit the new value (③). But on level zero no modifications to the MBR *L0A* are necessary. Now the insert without a split is complete.

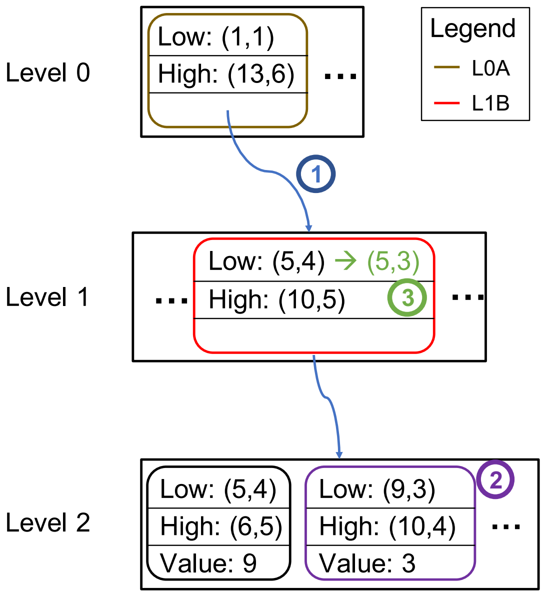
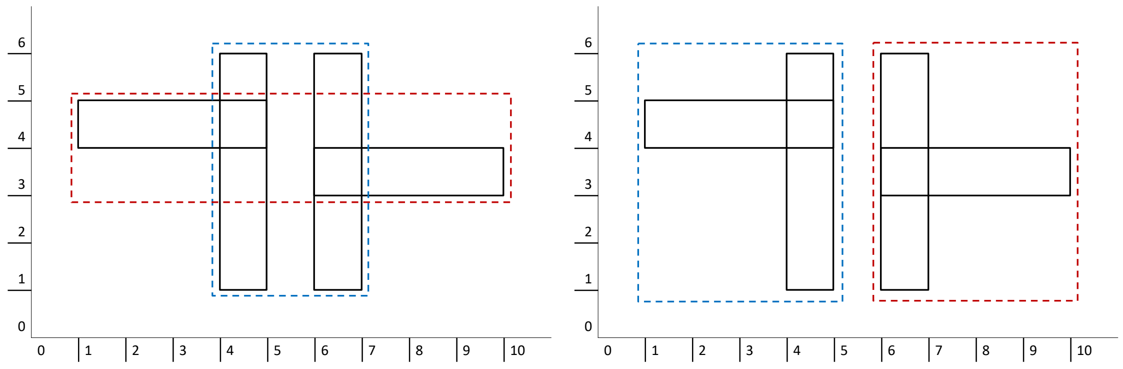


Figure 6: Steps of inserting value 3 into the R-tree from Figure 4.

Before showing an example of an insert causing a split, we first define the steps of a split in theory. A split starts when an insert operation inserts an object into a leaf, which already holds M branches. The new node is created as a child of the same parent as the node at capacity. If the parent node is also at capacity, it splits as well. Given the $M + 1$ branches, the split algorithm needs to divide all branches into two groups representing the new nodes. The algorithm distributes the branches so that the new nodes have optimal MBRs.

As an optimal split, Guttmann defines a distribution into two nodes where no other distribution has a smaller sum of areas of the two MBRs. He reasons, the result of a split should make it unlikely that both nodes need to be examined in a subsequent search query. The smaller the area of an MBR is, the smaller the chances are that a search query covers its area. Therefore, the resulting MBRs should be as small as possible. However, this assumes all search queries are evenly distributed among the available geometrical space. If that is not the case, a split minimizing the MBR areas can contradict the goal of avoiding unnecessary accesses to nodes. Figure 7a shows an example of such a split situation. The first split distribution has a smaller sum of areas. But for some parts, the MBRs overlap. Overlapping areas pose a problem if the search space of a query only contains one or none of the depicted data objects. In that case, both need to be accessed and checked. A distribution of the identical rectangles, as visualized in Figure 7b, would not require unnecessary node accesses in such a situation. The small example shows



(a) Minimal area.

(b) Minimal overlap.

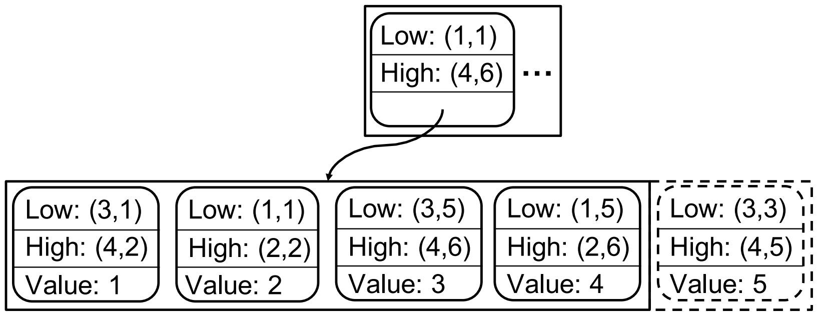
Figure 7: Example of a split distribution with optimization criteria.

that the optimal split depends on the query load.

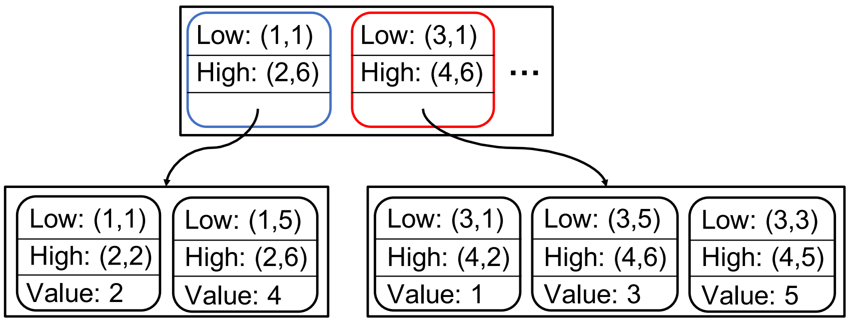
Calculating the distribution with the smallest sum of areas from all possible distributions has exponential time complexity. Therefore, the two most distant MBRs are selected and put into different groups. Greedily the MBR that would enlarge one of the groups MBRs the least is assigned to the respective group. The algorithm finishes if all MBRs are assigned or if one group has a size of $M + 1 - m$. All remaining MBRs need to be assigned to the other group. Otherwise, one of the nodes would have fewer than m entries. Experiments show that the exhaustive exponential-cost approach does not perform significantly better on subsequent queries.

Figure 8a shows an example with $M = 4$ and $m = 2$. An insert operation added the value 5 to the leaf, which already holds four data objects. Consequently, a split has to start and the branches need to be distributed into two groups. In this example, it does not make a difference if these groups are built to have a minimal overlap or minimal combined areas. The optimal distribution, shown in Figure 9, has both the minimal possible overlap, which is none, and the minimal sum of areas, namely 10.

After the algorithm distributed the branches into the groups, the split routine places them in two nodes and adds a new branch in the parent node. Also, the old branch is updated only to reflect the leftover values. Figure 8b shows the result of the split-leaf node.



(a) Insert of value 5 results in a split.



(b) Final result of the split.

Figure 8: Example of an insert causing a split.

3.3 Query

Querying an R-tree works similarly to querying any other tree structure. Starting at the root node, for each branch in a node, the query routine evaluates if the subtree might contain a match to the query predicates. Because an R-tree covers an n-dimensional space, queries need to define ranges or values to be found for each dimension. Additionally, data objects can be of non-zero size. Therefore, the search query needs to distinguish between geometrical relationships like overlapping, covering, containing, or equals.

Whereas only range queries in one-dimensional tree index structures might touch multiple leaf nodes, even point look-up queries can require looking at multiple leaf nodes in an R-tree. Since MBRs are allowed to overlap within a level, the query routine must consider all branches where the search space overlaps with the MBR. As a result, the query routine has to load all node branches into memory if the

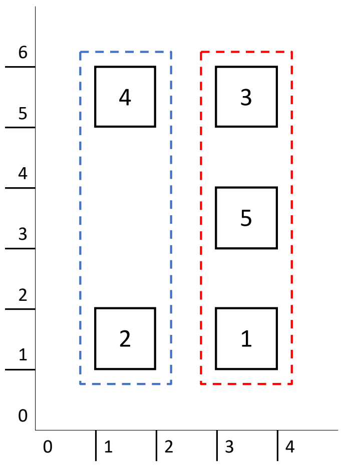


Figure 9: Optimal distribution of all values for the split of Figure 8.

node was selected as a candidate in the parent node.

Looking at the example in Figure 4, we observe this behavior if we query the tree for value 9. A query with the predicate to match exactly the dimensions $(5, 4); (6, 5)$ selects the *L0A* branch on level 0. On level 1, it first selects the branch *L1A*, because $(5, 4); (6, 5)$ lies within $(1, 2); (6, 6)$. After it queried the respective node on level 2, the query operation must continue checking the other branches on level 1. As we can see also the branch *L1B* contains the dimensions from the predicate. Depending on if the tree allows duplicates, the query routine either breaks after finding value 9 or continues.

On the other hand, if we query all values contained in the dimensions $(5, 4); (6, 5)$, the query touches only one leaf node. On level zero, the branch *L0A* is selected like in the other example. However, the leaf node under the branch *L1A* is skipped because the MBR has no overlap with the predicate dimensions. The branch *L1B* overlaps with the predicate dimensions of the query and is visited by the query. Assuming that no other tree branch has an overlap with this area, it is the only leaf visited by the query.

3.4 Delete

The delete routine of an R-tree starts by finding the leaf node containing the object to delete. Figure 10 shows an example of how to delete value 4. First, the delete routine queries the tree for the range of the value to delete (①). After the routine

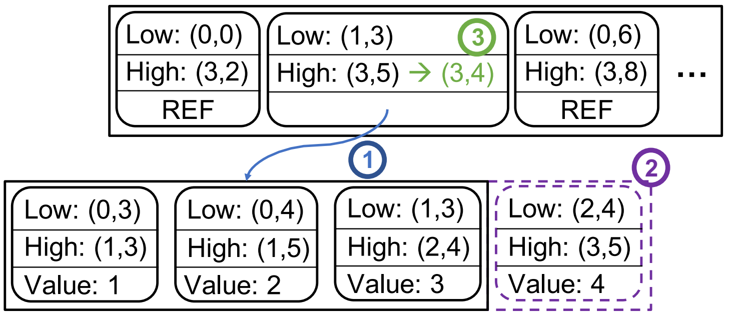
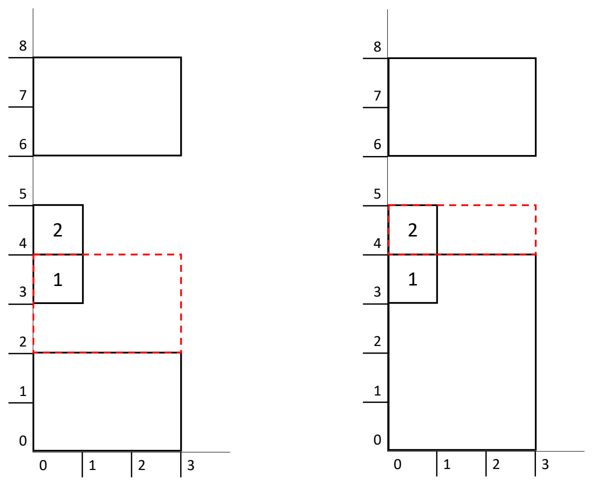


Figure 10: Steps for deleting value 4.

identified the right leaf, the corresponding branch gets deleted (②). Analogous to to the insert, the tree needs to be adjusted after the value was deleted. If the leaf still has more than m branches, the routine starts from the leaf and updates the MBRs until one MBR needs no further update, or the root is reached. In this case, the high point of the parent needs to be updated (③). The routine does not need to update an MBR if each boundary equals a child's MBR boundary in one dimension. Since all parent MBRs are guaranteed to be minimal, the routine does not need to continue if the break condition is satisfied on one level. If the leaf node has fewer than m branches after the delete, the merge routine starts from that leaf.

Analogous to the split, a merge starts whenever a node has fewer than m branches. Unlike the related B-Tree family, the R-tree does not merge the two nearest nodes but reinserts all branches at the root. For one thing, the nearest node is not clearly defined and finding a suitable candidate poses the same challenges as finding a good split distribution. For another, by reinserting the branches, the structure of the R-tree gets refined. If the reinsertion uses the same algorithm to chose subtrees as the insert routine, the tree structure is guaranteed to be at least as well balanced as it was before. In the best case, the structure is even more balanced since the insert algorithm has more branches available to choose the most suitable one.

As an example, we take the tree at the end from the delete example, shown in Figure 10. Only that now another delete operation deleted value 3. As a result, the leaf node now has fewer than m branches. The split routine reinserts all remaining branches one after another in other nodes. For a minimal total area, it would make sense to add value 1 to the left sibling $((0,0);(3,2))$ and value 2 into the right sibling $((0,6);(3,8))$. But since the values are added one after another, the area enlargement for the left sibling is smaller when adding the second value, as Figure 11a and 11b show. After the merge routine reinserted all branches, the underflow node can be deleted and its reference can be removed from the parent



(a) Reinserting the first value.

(b) Reinserting the second value.

Figure 11: Example of merging an under flown leaf with its siblings.

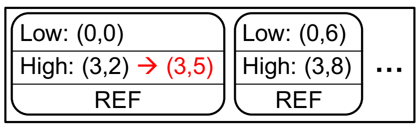


Figure 12: Final state of the parent node.

node. Figure 12 shows the final state of the parent node.

3.5 Related Work

Many major database systems implement R*-tree (e.g., boost [18], Oracle [34], IBM [26], PostgreSQL [19], and MySQL [44]). R*-tree improves the distribution of MBRs during insert and split compared to the original R-tree routines [3]. A subtree on the leaf level is selected during an insert based on the smallest overlap increase for the MBRs of the node. Subtrees on the inner levels are selected based on the smallest area increase among all branches, like the R-tree. For splits, R*-tree chooses a distribution having a small overlap and margin at the same time. Their experiments show that the decreased overlap leads to better performance for subsequent queries. Since splits require a lot of writes and are therefore a costly operation, R*-tree uses forced reinsertion. If a new value is added to a node at capacity, another value of the node is reinserted. Experiments show that in most cases, another node is selected for the reinserted value. Not only does this defers splits, but it refines the structure of the MBRs.

Hilbert R-tree reduces the number of splits as well, compared to the original R-tree insertion routine [32]. The nodes are structured as R-tree nodes, but each branch has additionally a Hilbert value assigned. The Hilbert value denotes the largest value of the subtree on the Hilbert space-filling curve. During insert, branches are not selected based on their MBR but based on their Hilbert value. As with B*-Tree, insertions to a node at capacity do not necessarily cause a split. The largest value is moved to the sibling node having the next larger value on the Hilbert space-filling curve. If the next two siblings are as well at capacity, the branches of the three nodes are distributed to four nodes. During a search operation, the MBRs are used in the same way as with the original R-tree search routine.

For static datasets, packing allows building an R-tree with improved space utilization [31, 35, 48]. A defined ordering function initially sorts all values. Afterwards, the values are distributed among the nodes. Either the nodes are filled in the order of the sorting, or the sorted values are sliced into smaller distributions, which are again sorted before the nodes are assigned. While a filling degree of 100% can be achieved, the MBRs can overlap, leading to poor query performance.

FOR-Tree [30], FAST-Rtree [50], and eFIND [8] were proposed as adaptions to flash-based storage technologies. Both use techniques to reduce the number of random writes to the storage medium. They keep changes in an in-memory memory table and use a hash table to identify nodes with changes in the buffer. FOR-Tree additionally uses overflow nodes instead of splits if a value is inserted into a node at capacity. An overflow node is only referenced by the original node and needs to be scanned as well during a query operation.

3.5.1 Failure-Atomic Byte-Addressable R-Tree

R-trees have not been an extensive subject of recent research in the field of PMem. So far, to the best of our knowledge, only one version of the R-tree was proposed, which adapts to the characteristics of PMem. In this section, we present FPR-Tree. Cho et al. propose it as an R-tree that guarantees crash consistency but leverages the byte-addressability of PMem simultaneously to offer high performance [13].

The authors chose to design a fully persistent R-tree storing all nodes in PMem. Like Dash, the FPR-Tree only relies on 8-byte word atomic stores by the CPU and adapted operation sequences to guarantee crash consistency [37]. For each modifying operation, all store instructions happen as a sequence of 8-byte writes, of which none leaves the R-tree in an inconsistent state. MBRs are allowed to be partially inconsistent after a crash. Because an MBR is too large to be updated atomically. However, the sequence of steps for an R-tree operation guarantees that a partially updated MBR does not produce incorrect results.

All nodes have a fixed-size array of branches that is neither sorted nor consecutively filled. A bitmap indicates for each node, which of its branches are currently valid. They limit the bitmap size to 56 bit to have 8 bit for a version number. Both need to fit in 8 B in order to be atomically updated in one instruction. So each node can hold at most 56 branches. For one thing, the version number is used for concurrent access and, for another, it is an indicator if the node is reorganizing. Both need to be updated in the same instruction since reorganizing operations need to indicate their completeness and persist their changes in one atomic operation. As a result, all of their changes are either persisted entirely or not at all. This allows a split to happen in place without a copy-on-write. The splitting operations copy the respective branches to the new node and only invalidate them in the old node when they are persisted in the new node. A sequence of *clwb()* and *sfence()* guarantee persistence. Inserting a value works similarly. After the value and its bounding box are persisted, the 8-byte word containing the bitmap is updated atomically.

Only the update of MBRs works differently. Even if all coordinates of the rectangle are encoded as single-precision float values, a rectangle takes up 16 bytes. Therefore, it could be possible that a crash leaves a partially updated MBR behind. The FPR-Tree accepts that as transient consistency since the implications do not invalidate the correctness of the R-tree. First, during the insert routine, an MBR only gets enlarged. If the MBR updates happen before the value is persisted to the leaf, a failure leaves an R-tree behind where its bounding boxes are not minimal anymore but still cover the whole data space. During a split, the update of the MBRs in the parent node happens in a sequence, which could lead to a situation where the MBR of the split node still covers the whole old sub-tree data space, but the new branch is already added. Still, consequent operations after a crash would not find duplicated entries since the child nodes are still marked as splitting.

After a restart, the whole tree must be scanned to detect nodes that are still locked or splitting. Otherwise, an operation could not detect if a node is locked or splitting from another running operation or an operation before a restart. If a node is found that is only locked, it needs to be unlocked. No matter which operation locked the node, its changes are either persisted entirely or not. But if a node in the splitting stage is found, the recovery process needs to complete the split. From the state of the node, it is possible to identify, which steps of the split were already persisted.

4 PeaR-Tree Design

PeaR-Tree is a failure-consistent, persistent and concurrent R-tree optimized for the use in PMem. Achieving persistence of an R-tree through the use of PMem poses different challenges than for a persistent R-tree on another storage medium. The byte-addressability requires different design decisions in terms of performance and correctness than for a block-based storage medium.

In this section, we present and motivate the design and implementation of PeaR-Tree as a failure-consistent, persistent and concurrent R-tree. First, we lay out the architecture together with implementation details in Section 4.1. Second, we discuss the adaption of the R-tree operations to PeaR-Tree in Section 4.2. Third, we introduce our recovery concept for persistent nodes in Section 4.3.

4.1 Architecture

In this section, we present the PeaR-Tree architecture. Our primary design goal is to build a failure-consistent and persistent R-tree. Further, we design PeaR-Tree to utilize available DRAM memory for an improved operator performance. Because DRAM is a resource with high space capacity costs, we allow limiting the DRAM consumption of PeaR-Tree. The architecture design incorporates dynamic memory placement. While failure-consistence is the most important aspect for all design decisions, we reflect the unique performance characteristics in the design of the components.

We describe the PeaR-Tree architecture from two perspectives. First, we present structural properties of a PeaR-Tree as an R-tree in Section 4.1.1. Next, we explain in Section 4.1.2 the design choices for the R-tree nodes in PeaR-Tree. Finally, Section 4.1.3 describes our allocation concept.

4.1.1 Tree Structure

A PeaR-Tree instance consists of node objects which are linked to a tree structure, as the example in Figure 13 demonstrates. They alone represent the current state of the instance. PeaR-Tree does not use any additional logging structure. To guarantee the persistence of all inserted entries, PeaR-Tree stores the nodes of the lowest level, the leaf level, in PMem. Nodes of other levels can be placed either in

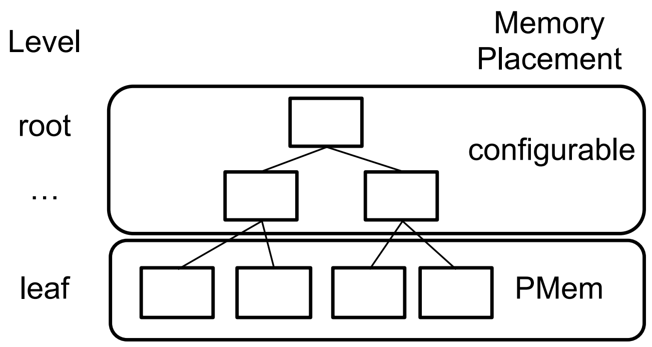


Figure 13: PeaR-Tree example structure.

PMem or DRAM. While they can change their placement during run-time, they are never duplicated to both DRAM and PMem at the same time. We explain the different persistence concepts in Section 5.

Due to the persistence property of PMem, the data is available in the same state after a system restart as it was before the restart. Thereby, it does not matter if the system was restarted because of a failure or after a clean shutdown. PeaR-Tree guarantees that all nodes stored in PMem are in a consistent state when the system restarts. A recovery process only needs to reconstruct the nodes of the tree that were not stored in PMem. All nodes stored in PMem do not need a dedicated recovery upon restart. However, an operation before the restart might not have finished updating a persistent node. Therefore, we introduce recovery states. All PeaR-Tree operations have a defined sequence of recovery states. Recovery states define a state of a node in which all information is stored consistently, but it needs recovery before an operation can use it. Each operation is designed to guarantee failure consistency within a recovery state and between the transition to the next state. Failure consistency guarantees that all information is stored correctly also in case of a failure. Should a node be in a recovery state after a restart, the next operation accessing it will detect the recovery state, as we explain in Section 4.3. Each recovery state can be uniquely identified and has a defined recovery process.

4.1.2 Node Design

Our primary design goal is to allow failure-consistent operations. In addition to that, we structure the node to reflect the performance characteristics of PMem. Existing persistent index structures show that it is beneficial to align node objects to the internal PMem cache line size of 256 B [5, 36, 37]. Access to PMem of sizes that are not a multiple of 256 B occur with an amplification. We structure the members of a node to reflect common access patterns within R-tree operations. Writes to PMem have a significantly higher latency than writes to DRAM. Therefore, we design the concurrency control and failure-consistency mechanisms to work with little metadata.

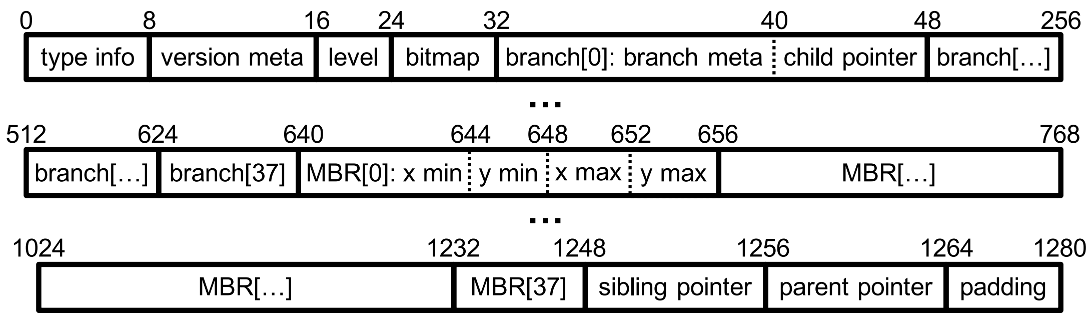


Figure 14: *PNode* memory layout.

Figure 14 shows the memory layout of a persistent PeaR-Tree node, the *PNode*. The size of all members is 1264 B and we use a padding of 16 B to align the size to 256 B. PeaR-Tree sets M to 38 and m to 12 (with M = maximum node size and m = minimal node size). Our experiments show that these values achieve the best performance.

The first 8 B encode the type of the node. Nodes that lie in PMem, the *PNode*, have a different type than nodes stored in DRAM, the *VNode*. Because of the type alignment, the type information occupies 8 B.

After the type information, a version meta number is stored for each node. It is used for concurrency control and to identify recovery states. The first bits are reserved to lock the node and as indicators for recovery states, as Section 4.2 explains in more detail. All other bits encode two version numbers. One continuously increases after each modification of the node, the other stores the current global version [37]. The continuously increasing number is used to detect concurrent modifications of the node while reading it. After each restart, the global version is increased. Whenever a node lock is acquired, the current global version number is written to the version meta number. With this mechanism, nodes that are locked from before a restart are detected. All of this information is stored within the same 8 B to allow an atomic update. Larger sizes are not guaranteed to be atomically written.

The next 8 B store the level number of a node. In the following, we label the root level as level zero. All levels beneath are numbered in ascending order. Consequently, if a root splits, the new root becomes level zero, and all other level numbers are increased by one. For the implementation, we abstract this concept and label the leaf level as level zero. Therefore, not all level numbers in the node need to be changed after a root split.

Next, the branch information is stored. All information is stored in two fixed-size arrays. Since the maximum number of branches is fixed, we can preallocate the space for all branches. We separate the MBR information from the concurrency control meta and the child pointer of a branch. All R-tree operations that perform a search for leaf nodes on the branch need to compare all children's MBRs before selecting the next child to query. Consequently, the MBRs are accessed frequently all one after another. Because they are stored in a contiguous memory region, less data needs to be loaded during these operations and more cache hits happen. In addition, the node is designed so that the first MBR is aligned to 64 B bit. This allows the compiler to use special AVX-instructions to compare multiple MBRs at once. A bitmap indicates which positions of the arrays are currently valid. With the 8 B bitmap, a value can be deleted atomically with a single instruction by unsetting the bit at the respective position.

Each node stores a pointer to its sibling node. If a node splits, the new node becomes the sibling node and the current sibling becomes the sibling of the new node. Therefore, the sibling pointers form a linked list of all nodes of a level. The list is needed to recover partially persisted index structures, as we explain in Section 5.1.

At last, each node stores a pointer to its parent node. It is set when a branch is added to a node. This implies the situation when existing branches are copied to a new node because of a split or merge. While the updates occur with an overhead of additional write operations to random locations, the reference information reduces the recovery time and allows a more lightweight concurrency control. The benefit of the parent pointer information for the recovery of a partially persisted tree structure is explained in Section 6.3.3. We discuss the relevance of the parent pointer for concurrency control in Section 4.2.1.

The *VNode* node type is structured similarly to the *PNode* node type, as we see in Figure 15. Although *VNode* objects are not stored in PMem and therefore do not need to be optimized for it, the similar structure makes the process of moving a node from DRAM to PMem less complex. We describe the details of that process in Section 5.1.2. The only difference of the *VNode* structure is that it does not use a bitmap. Branches in a *VNode* object are stored continuously in the first position of the array. The count member indicates how many positions are valid.

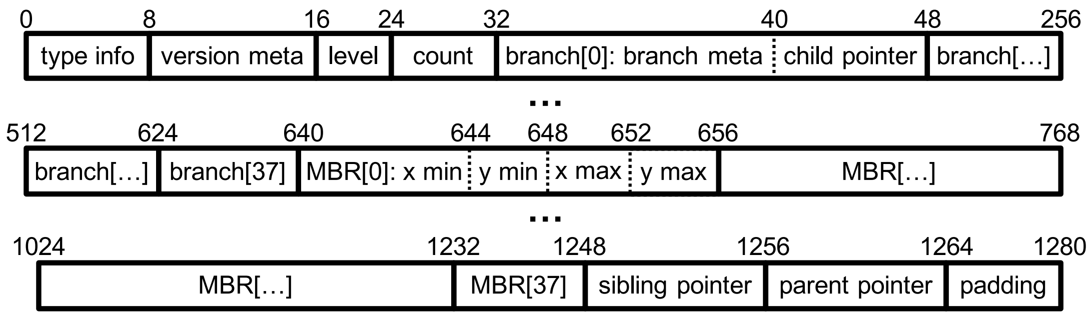


Figure 15: *VNode* memory layout.

If a branch is deleted, all branches after the position are moved. Write operations to DRAM have a lower latency, reducing the cost of the reorganization process. Moreover, the reorganization does not have to happen in an atomic 8 B write instruction. If a failure happens in between, the incomplete delete is not stored persistently.

4.1.3 Allocation concept

Current PMem aware libraries, like *PMDK* and *Memkind* already provide DRAM-like allocation interface. However, they introduce significant overhead, as described by the FP-Tree authors [45]. As with other existing systems, we preallocate both memory types in big chunks so that both allocations can be measured comparably [36, 37, 45]. While we preallocate enough memory for our measurements, PeaR-Tree is capable of dynamically allocating new chunks of memory. With preallocating the memory in chunks, the systems assume the same address range after a restart. A static address range allows normal 8 B pointers for all objects regardless of their memory location. Therefore, we do not differentiate the pointer types. However, we identify a persistent pointer by comparing its address to the persistent memory address range. Currently, PeaR-Tree is not capable of memory deallocations. FP-Tree uses amortized allocations, which allow reusing deallocated ranges in constant time complexity. So, adding a deallocation functionality would not introduce a performance overhead.

Name	**Identification**	**Description**
State 01	Node is leaf node $\wedge$ Node locked	Single branch modification in progress.
Insert 02	Reorganizing bit set $\wedge$ M Bits set in bitmap	Split started, no changes in parent node made.
Insert 03	Reorganizing bit set $\wedge$ Split Started bit set	Split calculated. Splitting of branch in parent node started.
Insert 04	Node locked $\wedge$ Split Started bit set	Splitted node copied values into this node. Splitting of branch in parent node started.
Insert 05	Node is inner node $\wedge$ Node is locked $\wedge$ (One child is in *Insert 02* $\vee$ One child is in *Insert 03*)	One child node is splitting. Update of branches has started.

Table 1: Insert recovery states with identification and description.

Name	**Identification**	**Description**
State 01	Node is leaf node $\wedge$ Node locked	Single branch modification in progress.
Delete 02	Node locked $\wedge$ Reorganizing bit set $\wedge$ Fewer than m Bits set in bitmap	Node is being merged because it has too few branches.
Delete 03	Node locked $\wedge$ Merging bit set	Node is being merged because a sibling node has too few branches.
Delete 04	Node locked $\wedge$ One child is in *Delete 02* $\wedge$ One child is not in *Delete 03*	One child node has too few branches. Merge candidate was not selected yet.
Delete 05	Node locked $\wedge$ One child is in *Delete 02* $\wedge$ One child is in *Delete 03*	Child nodes are being merged. Merge might have copied branches.
Delete 06	Node locked $\wedge$ Reorganizing bit set $\wedge$ Bitmap is zero	Branches of the node were merged with a sibling node.
Delete 07	Node locked $\wedge$ One child is in *Delete 06*	Merge is complete, but empty node is not deleted yet.
Delete 08	Node locked $\wedge$ No child is in *Delete 02*, *Delete 03*, or *Delete 06*	Merge is completed. All children have at least m branches.

Table 2: Delete recovery states with identification and description.

4.2 R-tree Operations

PeaR-Tree supports all operations the initial R-tree defines [22]. While the initial proposal does not define failure consistency or concurrency control mechanisms, database systems implemented them as part of their system-wide concurrency control and recovery protocols. I.e., Postgres implements the R-tree as a *Generalized Search Tree* (GiST) [24]. GiST is an index tree data structure that handles crash recovery and concurrency control for all operations and entries. Via an API, the operations for any datatype are implemented by the user. IBM Informix uses the R-link tree to allow concurrent access and an internal recovery protocol [27, 41]. However, all of these implementations assume a block layout on the persistent storage medium. Hence, we implement our own concurrency control and ensure failure consistency for PMem.

This section demonstrates the concurrency control mechanism and shows how we ensure failure consistency for each operation. We use fine-grained exclusive locking on branch and node level for all write operations and lock-free reading to allow lightweight concurrency. Each branch and each node has its own lock. As we define in Section 4.1.2 the lock is part of the version meta of a node or branch. Since the version meta has a size of 8 B, it can be updated atomically with a *compare-and-swap* (cas) instruction. The cas-instruction ensures that only one thread sets the lock bit if multiple threads try to set it at the same time. After a lock was acquired successfully, a thread fence guarantees that all changes are visible, which were made by other threads prior to this point [14, 15]. Another thread fence complements it before a lock is released. As we state in the intro of this section, our design guideline is to guarantee failure consistency with as little runtime performance overhead as possible. Therefore, we do not use any kind of logging structure or copy-on-write mechanism. Instead, we define a sequence of recovery states for each operation. For each recovery state, it is guaranteed that no information is lost in case of a failure. Each recovery state is uniquely identifiable from the members of a node. Tables 1 and 2 show all recovery states for the *Insert* and *Delete* operation. The PeaR-Tree *Query* operation does not modify the instance. A failure during a *Query* operation does not leave an incorrect state behind. Consequently, it does not need a sequence of recovery states. In the following, we explain why each state is failure-consistent and how the transition to the next state happens atomically. If each state is failure-consistent and the transition happens atomically, the whole operation is failure-consistent.

4.2.1 Insert

As we explain in Section 3.2, the R-tree *Insert* operation recursively selects a branch on each level until it reaches a leaf node, where it inserts the value. If the *Insert* operation of an R-tree inserts the new entry into a leaf, which has fewer than M branches (with M = maximum node size), no node is split. However, if the node holds M branches, the leaf has to split. The split can trigger a split in the parent node again until the root node splits and the tree structure grows in height. PeaR-Tree follows the same logic as the proposed procedure from R-tree [22] and R*-tree [3]. To guarantee failure consistency at any point in time during the operation, adaptions are made. They are based on the concepts from FPR-Tree [13]. In the following, we describe the details. As both scenarios differ in the complexity of their recovery states, we describe the failure consistency of the split and non-split cases separately.

Failure-Consistent Insert without Split

Figure 16 depicts a sequence diagram of the insert operation with its recovery state. Shown is the *PeaRTree* object issuing the insert, the root *PNode*, the *PNode* one level above the selected leaf node and the *PNode* leaf itself. An insert starts with selecting a path from the root node to the most suitable leaf node. The function *choose_subtree* recursively selects the best branch for a given value until it reaches the leaf level. Since no modifications are made during these calls, we omit the recursion from the graph for simplicity. As the *ChooseSubtree* algorithm, we use the original proposed R-tree *ChooseSubtree* algorithm [22].

The *PeaRTree* object locks the leaf node to guarantee the serialization of concurrent operations on the leaf node. That transitions the leaf node into recovery state *State 01*. It is identified by the node being a leaf node and being locked without any other status bits set. *State 01* does not differentiate if an *Insert* or *Delete* operation is in progress. The only change that violates the failure consistency within these operations is the locking. If a failure happens after any other change within this state, the node will still be in a consistent state after a restart, except it is locked.

Based on the returned path array, the *PeaRTree* object tries to acquire the branch locks bottom-up recursively until no further MBR update is needed. In this example, we assume all locks are acquired successfully. We deviate from the original R-tree insertion sub-operation order of updating MBRs after inserting the value to guarantee failure consistency, as we explain in the following. Acquiring the branch lock does not change the node's recovery state since it does not produce an

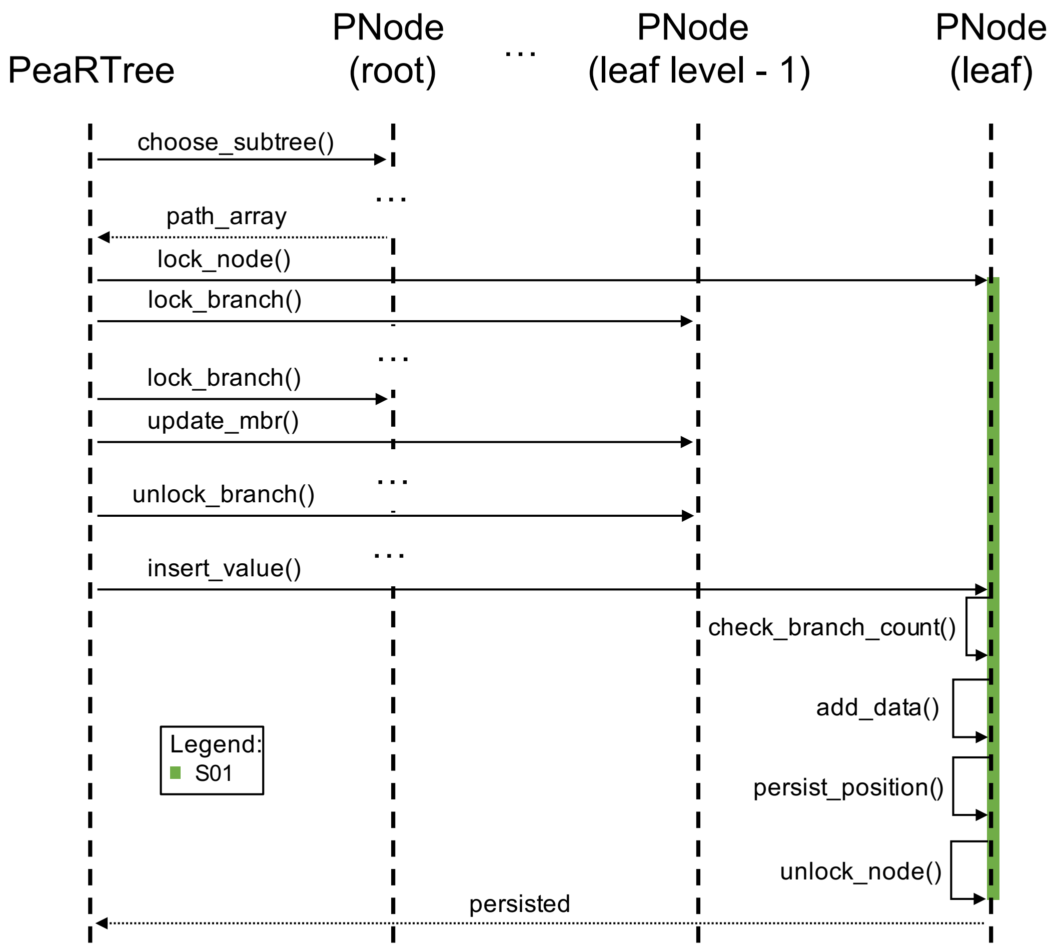


Figure 16: Sequence for inserting a value without causing a split.

incorrect state. Whenever any lock is acquired, 32 Bit of the 64 Bit atomic version are set to a global version [37]. The global version increases only during start-up. If a failure happens directly after acquiring the lock, subsequent operations after a restart find the branch locked with an old global version. It is guaranteed that the operation, which acquired the lock, is not running anymore since the global version is smaller than the current one.

Updating the MBRs of a branch does not result in a recovery state either. The updates only enlargen the MBRs. A failure after the update would result in a tree with bounding rectangles that are not minimal anymore, but still cover all subtree elements. Because an MBR is larger than 8 B, a failure could happen during the update, leaving a partially updated MBR behind. Still, the tree would be in a correct state. A point of the rectangle has a size of 8 B and is updated atomically. If an MBR is partially updated, it means only one of its points was changed. As with a failure after the update, this only enlarged the MBR. The MBR with one updated point still covers all children. Hence, all MBRs updates are treated as changes that do not require recovery. After all MBRs are updated, the branch locks are released.

For the insert to finish, the value needs to be added to the leaf node (*add_data*). As this scenario assumes no split is required, the data can be added at a free position in the respective arrays of the *PNode* object. After the writes are persisted, the corresponding bit is set in the bitmap with *persist_position*. This write operation does harm the failure consistency, as it is an 8 B write. After the bitmap is updated, the lock on the node is released, which ends the recovery state.

Failure-Consistent Insert with Split

If an insert requires one or multiple splits, the procedure shown in Figure 16 deviates at the end. In the function call *insert_value* from the *PeaRTree* to the *PNode* leaf object, a check detects if the leaf needs to split (*check_branch_count* in Figure 16) . Should a split be required, the leaf is first split before the new value is added and persisted. Figure 17 shows the PeaR-Tree split procedure starting from the *insert_value* function call from the *PeaRTree* object. In addition to the function calls, each node shows its recovery states over time. The order of steps is the same as with the FPR-Tree in-place split procedure [13]. The advantage from the FPR-Tree in-place split procedure compared to other R-tree split procedures is that it is failure-consistent.

As the first step, the reorganizing bit is set in the node version meta to indicate that the node is splitting. With this change, the node transitions into recovery state *Insert 02*. Its identification is that the node is locked, the reorganizing bit is set and the bitmap's first M bits are set (with M = maximum node size). After the change is persisted, *create_distributions* calculates the branch distribution into two groups. The groups denote which values are copied to the new node and which stay in the split node. We use the R*-Tree split algorithm. The split is calculated on all branches of the overflown node and the newly added value and saved in temporal data sets. In the next step, the overflown node acquires a new node from the *Allocator*. Apart from handling the memory allocation, the *Allocator* saves the pointer in a persistent buffer. For the point in time, no persisted reference to the newly acquired node exists elsewhere. A persisted memory leak would occur if no reference was persisted in the case of a failure.

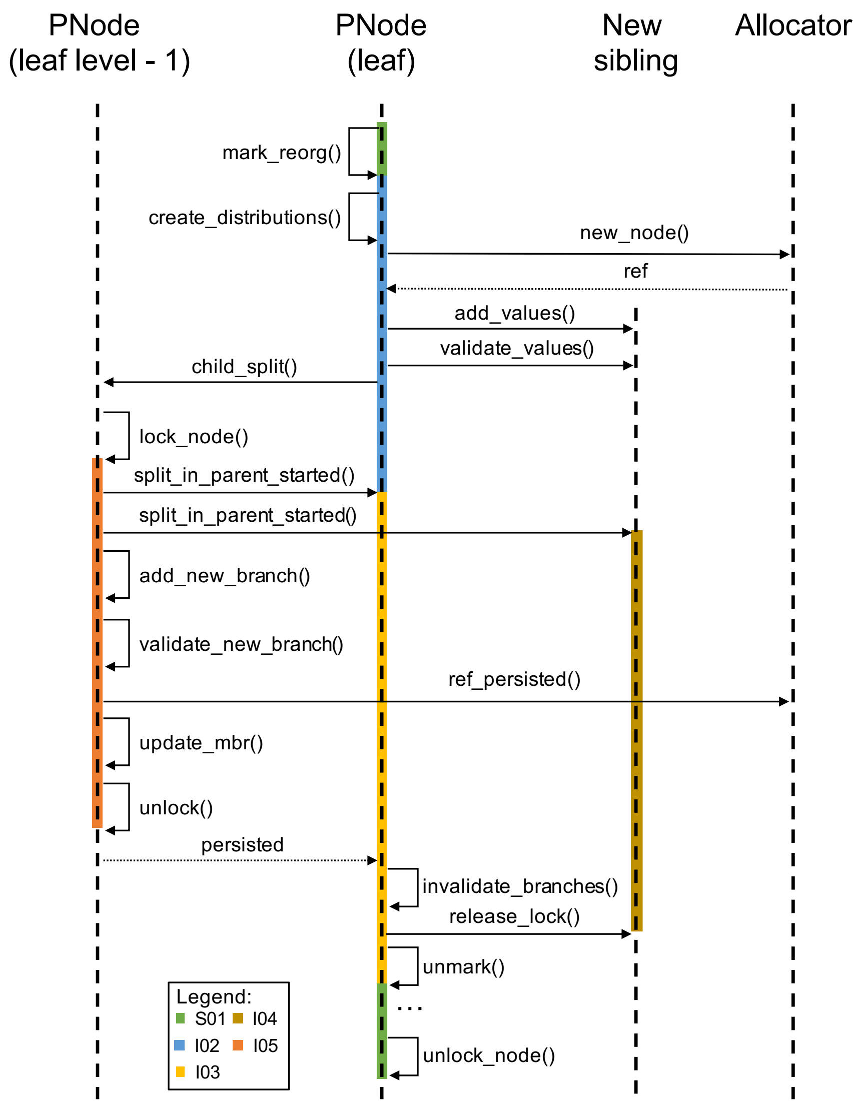


Figure 17: Sequence for inserting a value causing a split.

The new node is created locked but in no recovery state. As long as the branches still exist in the overflown node, there is no need to recover the incomplete new node. One after another, the branches determined to be in the new node are copied. Only after all branches are persistently copied, the bitmap of the new node is set to validate all copied branches with *validate_values*.

Now that the new node is filled with the branches, it can be added to the parent node. After the parent node acquired its own lock, it is in recovery state *Insert 05*. For a recovery process, it indicates that a reorganization in subnodes is in progress. To identify which kind of reorganization is happening and its progress, the recovery process has to look at all child nodes. For that reason, the parent node sets another status bit in the node version meta of its splitting child and the new child by calling *split_in_parent_started*. It transitions both nodes into the next recovery state respectively, the states *Insert 03* and *Insert 04*. If a failure happens between locking the parent and setting the bits, a recovery process can still identify the pair by comparing the branches with each other. The next step for the parent node is to add the new branch and validate it in its bitmap. Should this require a split of the parent as well, it works the same way as for a leaf node. Before the split of the leaf node continues, the split of the parent node is persisted. As the steps are the same, the recovery states of the parent split are the same and we do not explicitly cover them further. As soon as the reference to the new node is persisted, it can be deregistered at the *Allocator* since there is no risk of a persistent memory leak anymore (*ref_persisted*). As the last step in the parent node, *update_mbr* updates the MBR of the overflown node accordingly to the new values. In this case, the MBR is likely to decrease in size. Still, it is no problem for a recovery process if the failure happens so that only half of the MBR gets updated. It can still identify the splitting branch in the parent node and recalculate the MBR. Finally, the parent node unlocks itself.

The overflown node invalidates all entries, which were copied to the new node. Assuming the split calculation assigned the new value to the overflown node, the lock on the new node is released. The split for the new node is complete, and it is in no recovery state anymore. The overflown node transitions back into recovery state *State 01* by resetting the status bits to complete the *Insert* operation. Then, the data is added and persisted as with the *Insert* without a split in Figure 16. Finally, the node is unlocked and in no recovery anymore. The *Insert* operation has finished.

Concurrent Insert

If an *Insert* operation is executed concurrently with other operations, the operations are likely to access the same nodes. We explain how all updates from the *Insert* operation and other concurrent operations are serialized correctly.

While the insert path is recursively selected and the value is not inserted, concurrent operations operate on the nodes along the path as they are not locked. A split or merge of a node that was already selected by the *Insert* operation invalidates its selection. If this happens before the *Insert* operation acquired the branch lock of the selected branch in the node, it has to redo the selection on the level above. PeaR-Tree detects such a case by storing the version number of a node upon the branch selection in the function *choose_subtree*. It is compared before acquiring the branch lock. If the number changed, branches of the node were moved, or a new branch was added. PeaR-Tree does not check which modification caused the change but reselects a branch in the parent of the changed node. The reselection starts in the first ancestor node along the path, which has not changed.

Before the branch locks are acquired, the lock of the leaf node is acquired. It is held until the value was inserted and ensures that only one operation at the time inserts values into the leaf. If the node is currently locked, the *Insert* operation waits until it is unlocked.

After the *Insert* operation has locked the selected branch in a node, the node cannot split or merge. When a restructuring operation like a split or a merge moves a branch, it first waits until the branch is unlocked. The *Insert* operation locks all branches bottom-up along the selected path for which an update of the MBR is necessary. In the case of a necessary reselection, as described above, previously acquired locks are rereleased. If no reselection is necessary, the MBRs along the path are updated.

All changes to the node along the insert path afterwards do not affect the correctness of the *Insert* operation. The changes to the MBRs are persisted and other operations reflect them regardless of whether the value is already persistently inserted in the leaf or not.

In the case of a necessary split, the parent pointers are used to propagate the split upwards in the tree structure. Whenever the *Insert* operation fails to lock its parent for the split propagation, it updates the pointer reference from memory again before the next attempt. Continuous updating is necessary to notice if another operation splits the parent and the node gets a new parent assigned. This operation updates the parent pointer for all moved branches in the respective child node.

In the case that the parent of the leaf node splits and a branch is locked, the *Insert* operation only continues under one condition. If the child node of the locked branch is marked as splitting without the *started reorganization in parent* bit set, waiting causes a deadlock. A concurrent *Insert* operation splits this leaf node and waits for the parent to unlock. Therefore, the *Insert* operation that acquired the lock of the parent node first finishes without waiting for the branch lock. If the branch is moved to a new node, it is created with the lock bit set.

4.2.2 Query

PeaR-Tree's *Query* works the same way as the R-tree *Query* operation described in Section 3.3. Failure consistency is not an issue for the *Query* operation as it does not modify the tree structure. To traverse the tree structure, the *Query* operation uses a recursive depth-first search algorithm. The *Query* itself does not acquire any locks but respects node locks hold by other operations. If a *Query* operation encounters a locked node, it restarts at the parent node. If the node is locked because of a split or merge, the branch selection in the parent is invalidated. In the case that a node splits or merges while the *Query* operation traverses a subtree of a node, all selections made in the subtrees of the node are invalid as well. The *Query* operation does not track which nodes it already queried. If a subtree is moved to a different node that is visited by the *Query* operation afterwards, entries are included twice in the result set. Therefore, before the *Query* operation returns from a recursive call back to the parent node, it compares the node version meta version to the initial version meta version. If the version changed, the *Query* operation erases all values added in the subtree from the result set and restarts the search in the parent node.

4.2.3 Delete

We modify the delete behavior of R-tree, as it was introduced in Section 3.4, for PeaR-Tree to merge nodes efficiently and failure-consistent at the same time. As FPR-Tree, we chose the in-place merge with one other sibling in contrast to the reinsertion merge as proposed by R-tree [13, 22]. A reinsertion requires explicit logging to atomically remove the entry from one node and add it to the new node. Similar to the *Insert* operation description, we first define the *Delete* operation sequence without a merge. Afterwards, we explain the in-place merge in detail.

Failure-Consistent Delete without Merge

For deleting a value, the tree issues a recursive depth-first search starting at the root with the function *remove*. Figure 18 shows the sequence of steps for a delete operation that does not cause a merge. The *PNode* leaf object on the right is the node containing the value to be deleted. In the first step, the delete operation locks the node. With this change, the node transitions into recovery state *State 01*. The state indicates that a change to the node is happening, but nothing needs to be recovered. In the case of the delete, this is true because the following change is made atomically. Finding the branch containing the value to delete does not

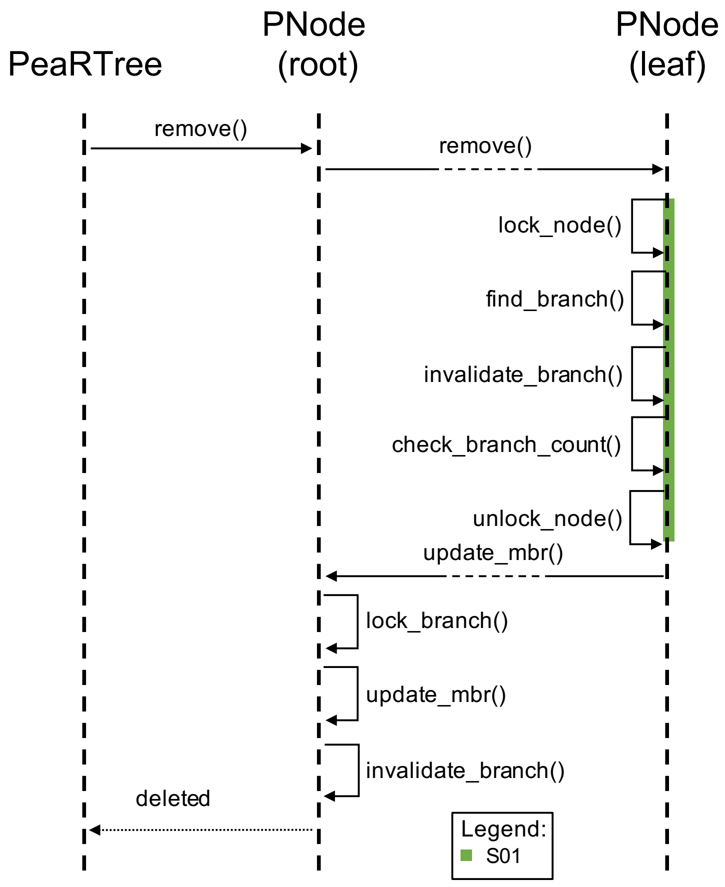


Figure 18: Sequence for deleting a value without causing a merge.

require any modifications. Only invalidating the entry in the bitmap with *invalidate_branch* requires a modification to a member variable of the *PNode*. However, the bitmap can be changed in an atomic 8 B write. Although the delete operation has not finished yet, the value is now persistently deleted. Subsequent operations regard the branch as invalid and do not read the value. *check_branch_count* tests if still enough branches are valid. In this scenario we assume, there are still enough valid branches.

The following modifications only restore the invariants of the R-tree but do not change the correctness of the tree. In the case of a failure, they consequently do not need to be restored. Recursively the tree gets updated bottom-up. On each level, the corresponding branch lock is acquired. Similar to the insert MBR updates, it does not change the recovery state of the node. The change in size of the MBRs does not change the correctness of the tree since it does not influence if values are found or not. All values that lie under an MBR and are not deleted can be found regardless of whether the bounding box is minimal or not. Furthermore, the deleted value cannot be found since the update only starts after the branch is invalidated in the leaf.

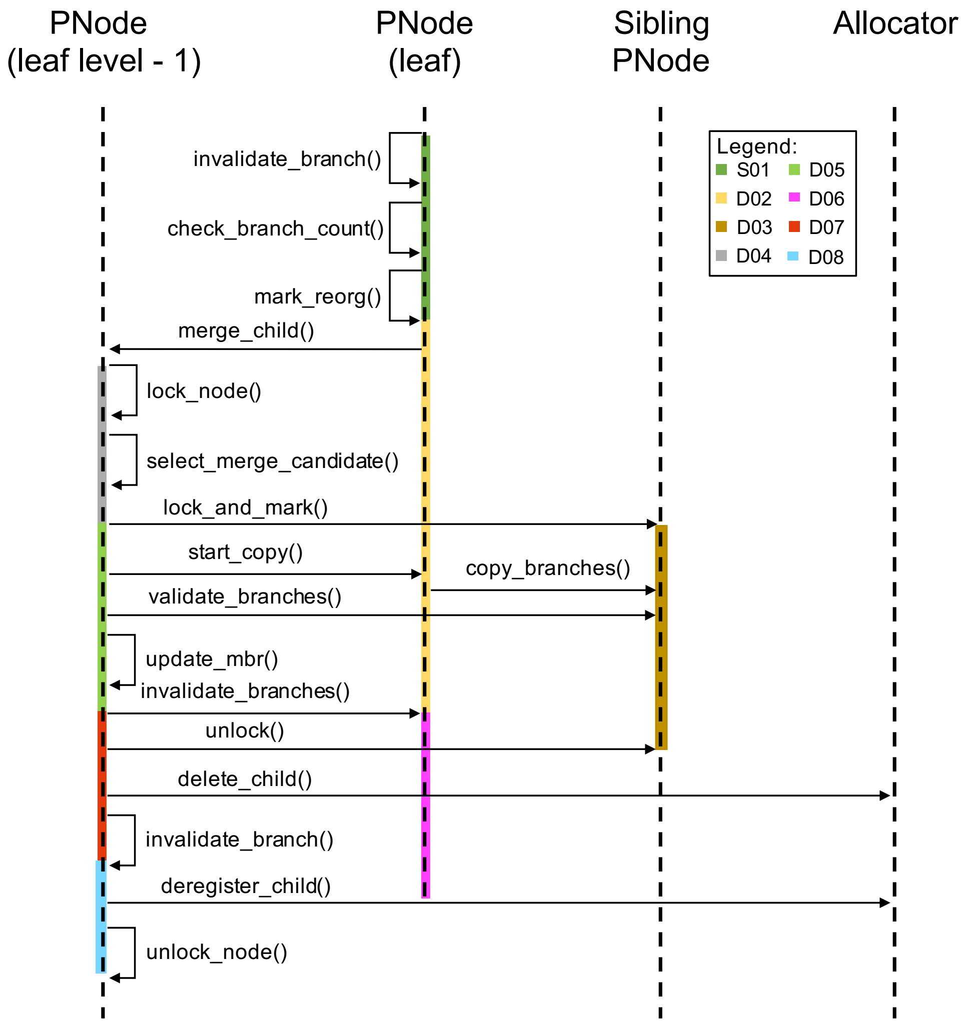


Figure 19: Sequence for deleting a value causing a merge.

Failure-Consistent Delete with Merge

Figure 19 shows the sequence of steps for a delete operation resulting in a merge. We omit the initial delete steps since they are the same as in Figure 18 (Page 42). After the value to be deleted is invalidated, the delete operation detects that the node has too few branches with *check_branch_count*. Instead of unlocking the node, the delete operation marks the node as reorganizing. It is the same bit in the node version meta, which is used during the split to indicate the reorganizing state. For that reason, recovery state *Delete 02*, which the node transitions to, is indicated by the set reorganizing bit and a bitmap where fewer than m bits are set ($m =$ minimal node size). This makes it possible for a recovery process to differentiate between a started split and a started merge.

The problem with the reinsertion merge for a recovery process is to detect if a value has been reinserted or not. A recovery process can detect if a node is in the merging process and which branch is currently being reinserted by checking the bitmap and the node version meta. However, it is not possible to atomically reinsert a value. Validating the value in the new node and invalidating it in the old node are two instructions. Since a failure can happen in between these two instructions, a recovery process needs to check for that case. Without additional logging, the recovery process needs to check the whole tree, whether the branch got reinserted or not. For that reason, we choose the in-place merge with a sibling.

After the node is marked as reorganizing, the node triggers with *merge_child* the parent to merge itself with a sibling. First, the parent locks itself and selects a merge candidate. As the merge candidate, the parent selects the child, which would need the least area enlargement to contain the left-over branches from the underflow node. Then, the child candidate gets locked and the merging bit is set in the node verion meta (*lock_and_mark*). Until the child candidate is not marked as merging, the parent node is in recovery state *Delete 04*. With marking the merge candidate as merging, the merge candidate enters the recovery state *Delete 03*. Simultaneously, the parent node transitions into recovery state *Delete 05*. A recovery process needs to differentiate if the merge already started or was just calculated. Consequently, the state is determined by an inner node being locked and having a child in *Delete 02* and another child in *Delete 03*. After the branches are copied, they are validated in the sibling node. This does not change the recovery state since a recovery process could not determine if the bitmap was already changed or not yet changed.

After all branches are persistently copied and validated, the parent node updates the MBR of the merge candidate. Only then are the branches in the node with fewer than m branches invalidated since they are now visible and findable in the other node. With all branches invalidated, the node to delete transitions into recovery state *Delete 06*. In this state, the bitmap is zero and the node is marked as reorganizing. The parent transitions into recovery state *Delete 07* since the node to delete is not anymore in *Delete 02* but in *Delete 06*. Before the delete operation removes the node to delete, the merge candidate node is unlocked. With unlocking the merge candidate node, it is again in no recovery state.

The deletion of the now-empty node happens in three steps. First, the parent node marks the node as deleting with the allocator. To prevent memory leaks, a reference to the node has to be kept persistently outside the tree structure. On the one side, the memory cannot be freed before it is guaranteed that no other operation on the tree will access it. Conversely, if the last persistent reference gets deleted before the memory is freed, a memory leak can occur in the case of a failure. Therefore, the delete operation invalidates the reference in the branch in the second step and permanently frees the memory in the third step. Invalidating the branch referencing the empty node transitions the parent node into recovery state *Delete 08* since no reference to a child in *Delete 06* is valid anymore. Finally, the parent node gets unlocked and exits the recovery state.

In this scenario we assume the merge candidate has enough space to fit the other branches. If the amount of valid branches of both nodes sums up to more than M branches, the branches are distributed with the split algorithm. The nodes are reorganized so that both nodes have at least m branches and at most M branches (with M = maximum node size, m = minimal node size, and $2 \leq m \leq M/2$). For the recovery states, the only difference is that the underflow node is not deleted. After the values have been copied and validated accordingly, the underflow becomes unlocked and the reorganizing bit is reset.

Concurrent Delete

Similiar to *Insert* operations, modifications of concurrent *Delete* operations are serialized with all other concurrent operations.

Before the value is found in the leaf node, no modifications are made. The search for the leaf uses the same algorithm as the concurrent *Query* operation. When the leaf is found, it is locked. Should the locking attempt not be successful, the *Delete* operation waits. While it retries to acquire the lock, it checks if the node meta version of the parent node changes. If the branch pointing to the correct leaf node is moved, the *Delete* operation detects the increased node meta version.

In the case of a necessary merge, the *Delete* operation follows the same approach as the split during a *Insert* operation. The merge is propagated with the parent pointer. If a parent is locked, its reference is continuously updated to detect a changed parent node.

After the merge finishes, the *Delete* operation updates the MBRs bottom up. For the update, all MBRs in the leaf node are scanned and the calculated MBR is compared with the MBR of the branch in the parent node. If an update is necessary, the branch gets locked and updated. Otherwise the *Delete* operation finishes.

After that, the next parent node is updated with the same procedure. Before and after the MBRs of a node are scanned, the node version meta is compared to detect possible changes in the branch structure. If the node has changed, the *Delete* operation retries.

4.3 Node Recovery

PeaR-Tree instances can be recovered regardless of whether the shut-down of the system happened because of a power failure or as a clean shut-down, where all operations could finish. Since only the *PNode* objects are persisted, a recovery process needs to reconstruct the corresponding *PeaRTree* object. After the recovery process constructed the object, it can directly accept user queries. Therefore, the recovery process needs to read the last used global version from a persisted metadata file, the address of the root node and the trees height.

As we describe in Section 4.1.3, we assume that the PMem address range stays the same after each restart. Therefore, we do not need to adjust pointers during recovery. Unless the clean-shutdown flag is set, the *PNode* objects need to be checked during runtime, as they could be in a recovery state. Whenever an operation finds a *PNode* object in a recovery state, a recovery process for that node object is started. The check if a node needs any kind of recovery process or not requires only reading the version meta. All recovery states include a set lock bit as an indicator. If a node is locked with a global version smaller than the current one, a node is in a recovery state. However, if the flag is set, no checks need to be made during runtime. Furthermore, before the next clean shut-down, all nodes must be checked once to ensure no node is in a recovery state. Only then, the clean shut-down flag can be set again.

Each recovery state can be identified uniquely. Depending on the recovery state of an operation, the recovery process either rolls back the changes or finish the operation. Table 1 and Table 2 list all recovery states and their identification properties. In the following, we describe the recovery steps for each state.

A *PNode* object in recovery state *State 01* indicates that a single branch modification did not finish properly. Either an *Insert* operation or a *Delete* operation did not unlocked the node. The recovery process does not distinguish between the two scenarios. It only unlocks the node. If the entry was already added or deleted before the failure, the change is persistent, although the operation did not return successfully. PeaR-Tree does not guarantee transaction management. The application using PeaR-Tree needs to ensure that operations that did not finish before a failure are either rolled back or continued. If a value was deleted such that the node has fewer than m branches, the recovery process still only unlocks the node. A node with fewer than m branches does not produce incorrect query results. Additionally, the next modifying operation resolves the problem. Insert operations add a branch so that the node would have m branches again. Delete operations trigger a split resulting in a merge for the node.

4.3.1 Insert Recovery States

If in a recovered tree an operation detects a *PNode* object in recovery state *Insert 02*, the split is not continued. Recovery state *Insert 02* can only be reached in leaf nodes. In this state, the value, which caused the split, has not been inserted yet. Therefore, the split is not required anymore. The recovery process only resets the reorganizing bit, unlocks the node, and the operation continues that discovered the recovery state.

If an operation finds a *PNode* object in recovery state *Insert 03*, it means the split in the parent node is finished. Otherwise, the operation would have found the parent node in recovery state *Insert 05* since the tree is always traversed top-down. Although the value, which caused the split, is not yet inserted, the recovery completes the split. Rolling back the changes and deleting the already created node would require more writes than finishing the split. The recovery process needs to find the sibling and identify the branches, which were copied. Since the sibling is not unlocked until the splitting node leaves recovery state *Insert 03*, it can be identified as the sibling node, which is marked as splitting. When the recovery process identified the copied branches, they are invalidated in the splitting node. Afterward, the sibling node is unmarked and the splitting node is unlocked and unmarked as well. If the bitmap of the splitting node already has fewer than M bits set, this last step is done directly. The nodes are now in no recovery state anymore and the operation can continue.

The recovery process for a *PNode* object in recovery state *Insert 04* works analogous to recovery state *Insert 03*. We know that the values are already copied and invalidated in the splitting node. Therefore, the recovery process resets the splitting bit in the found node and identifies the splitting node and recovers it. The splitting node is in recovery state *Insert 03*.

If an operation finds a *PNode* object in recovery state *Insert 05* the split is only continued if a child node in recovery state *Insert 04* can be found in a valid branch. Otherwise, the reference to the new node was not yet added and therefore not persisted. If it is not found, the recovery process unlocks the found parent node and reset the reorganizing bit and the splitting bit in the splitting child node. If the recovery process finds a child in recovery state *Insert 04*, the the recovery process finishes the split. Although a split is not necessary anymore, it is beneficial to use the already calculated distribution of the node. Reverting the split at that stage requires recalculating the distribution after the next insert to the node again. To complete the split, the recovery process updates the split MBR and then proceeds with the recovery process for recovery states *Insert 03* and *Insert 04*.

4.3.2 Delete Recovery States

For recovery states *Delete 02* and *Delete 04*, a recovery process also does not finish the merge for the same reason. An operation can only find a node object in recovery state *Delete 02* if the split had not started in the parent node. Otherwise, the operation would have found the parent node in recovery state *Delete 04* or *Delete 05* before. In recovery state *Delete 04* no merge candidate was selected yet since it ends with a child entering recovery state *Delete 03*. For both states, a recovery process only needs to unlock the nodes and reset the reorganizing bit.

If a node is in recovery state *Delete 05*, the recovery process does not finish the merge as well. Still, the recovery process has to check the child triggering the merge and the merge candidates branches. It is essential that no branch is valid in two nodes. With the merge candidate being in recovery state *Delete 03*, the copied branches could already be validated. However, the only way for the recovery process to determine this is by comparing both branch arrays against each other. Entries that are duplicates and valid in both nodes are deleted from the merge candidate.

A node in recovery state *Delete 06* cannot be discovered. Its parent in recovery state *Delete 07* is discovered first. For a node in *Delete 07*, the merge will be completed by the recovery process. If a child is still in recovery state *Delete 03*, it needs to be unlocked. All branches are guaranteed to be persistently copied. Otherwise, the parent node would still be in *Delete 05*. If the child in *Delete 06* has not been marked as deleted yet, the recovery process does that. Furthermore, it invalidates the branch pointing to the empty child. Finally, the parent node needs to be unlocked. This last step is also the only recovery step for recovery state *Delete 08*.

5 Adaptive Persistent Tree Index Structure

To achieve better runtime performance, existing persistent index structures use the concept of selective persistence [36, 45, 55]. With selective persistence, only the data is stored in PMem that is essential to recover the full structure. In a tree index structure, these are usually the leaf nodes. Everything else is stored in DRAM and is recreated upon restart. The recreation of the volatile parts during restart increases the start-up time. As a trade-off, the runtime performance is increased because DRAM has a better access performance. Usually, the amount of data kept in DRAM grows with the number of entries in the index with no upper bound. As a result, selective persistence is agnostic to the available amount of DRAM.

PeaR-Tree allows storing parts of the tree in DRAM as well. Since DRAM is more expensive than PMem PeaR-Tree offers a configuration limiting the DRAM consumption. For this, we introduce the concept of *adaptive persistence. Adaptive persistence* allows setting a limit for the DRAM consumption of an instance regardless of the contained number of entries. PeaR-Tree dynamically defines which parts of the tree are stored in DRAM and which are stored in PMem. For each node, PeaR-Tree determines if the node lies in DRAM or PMem. Thereby, no data is duplicated. It means that the node is stored either completely volatile in DRAM or persistently in PMem.

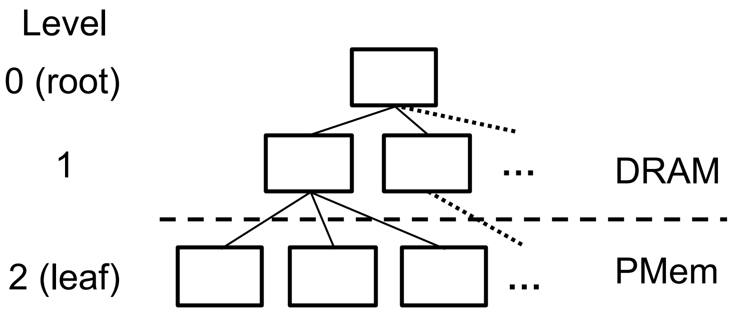


Figure 20: Example of a PeaR-Tree instance with *level-bound adaptive persistence.*

PeaR-Tree introduces two configurations of limiting the DRAM consumption, *level-bound* and *node-bound. Level-bound adaptive persistence* defines a maximum number of tree levels, which are stored in DRAM. *Node-bound adaptive persistence* defines a maximum number of nodes, which are stored in DRAM, regardless of the level. For both configurations, all leaf nodes lie in PMem no matter how much space in DRAM is left. Otherwise, a tree's recovery would not be possible because

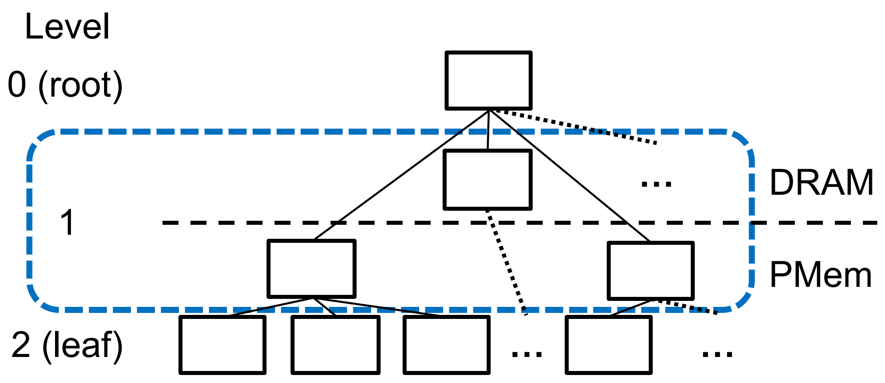


Figure 21: Example of a PeaR-Tree instance with *node-bound adaptive persistence*.

the values in the leaf nodes would not be persisted. The placement of the nodes in the levels above depends on the tree size and configuration. PeaR-Tree tries to place as many nodes as possible in DRAM to leverage the better access performance. With both concepts, the nodes closest to the root are placed in DRAM and all nodes below in PMem.

A *level-bound* PeaR-Tree instance stores the first X levels in DRAM, where X is the defined maximum number of DRAM levels. Figure 20 shows an example with a maximum of two DRAM levels. Since the maximum size of the upper levels of an R-Tree is independent of the contained entries, limiting the levels guarantees a limitation of the DRAM consumption. The nodes of level 0, the root, and level one are all stored in DRAM. In the example, all nodes of level two are stored in PMem. Both because it is the leaf level and because it is not one of the most upper two levels. If the root node splits and the tree grows in height, the nodes of level one, which is then level two, are moved to PMem. We explain this process in Section 5.1.

With *node-bound adaptive persistence*, as many nodes as the limit allows are stored in DRAM. The root node always lies in DRAM. PeaR-Tree keeps the nodes closest to the root node in DRAM so that as many complete levels as possible are placed in DRAM. As a result, only the nodes of one level are partially stored in DRAM and partially stored in PMem. We call this the hybrid level. Figure 21 shows an example where level one is the hybrid level. The first and third child of the root node are stored in PMem, but the second child is stored in DRAM. If a new node is created in DRAM, PeaR-Tree moves one node from level one to PMem. The details of this process are explained in Section 5.2.

Since at most one hybrid level at a time exists, it is guaranteed for all persistent nodes that all nodes of their subtree are persistent as well. This property allows recovering an adaptively persisted PeaR-Tree instance without reading all persisted nodes. In Section 5.3 we explain each step of the recovery process and analyze the complexity.

5.1 Level-Bound Adaptive Persistence

With *level-bound* persistence, levels of a tree are either completely persisted or not at all. Based on the configuration, the number of levels that lie in DRAM are fixed. The lower levels always lie in PMem and the upper in DRAM and the leaf level is always persisted. Whenever the root node needs to split, another level is added above the root node. Therefore, during runtime, levels need to be moved from DRAM to PMem not to exceed the maximum DRAM level count. In the following, we explain the concept of persisting complete levels and how the operation works failure-consistently and concurrent to other operations.

5.1.1 Persisting Concept

When the number of DRAM levels is fixed, each additional DRAM level causes another level to be moved to PMem. Splits of non-root nodes create their node with the same memory placement. After the root splits, the tree checks if too many levels lie in DRAM. If that is the case, the lowest DRAM level is moved to PMem. For a tree with a fixed number of one DRAM level, the first root node split works as shown in Figure 22. Because the leaf level always lies in PMem, only one level lies in DRAM after the first root split.

When the next root split happens, the new root is created in DRAM (see the first part of Figure 23). From the second part, we see that afterwards, all nodes from level one are copied one after another to PMem. For the duration of the copying

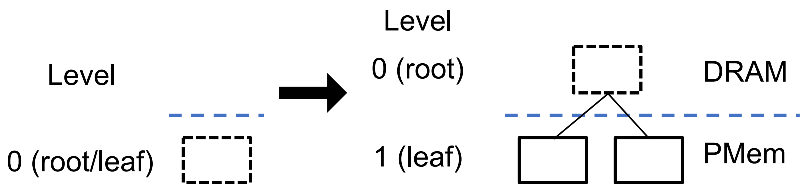


Figure 22: Initial root splits for *level-bound adaptive persistence*.

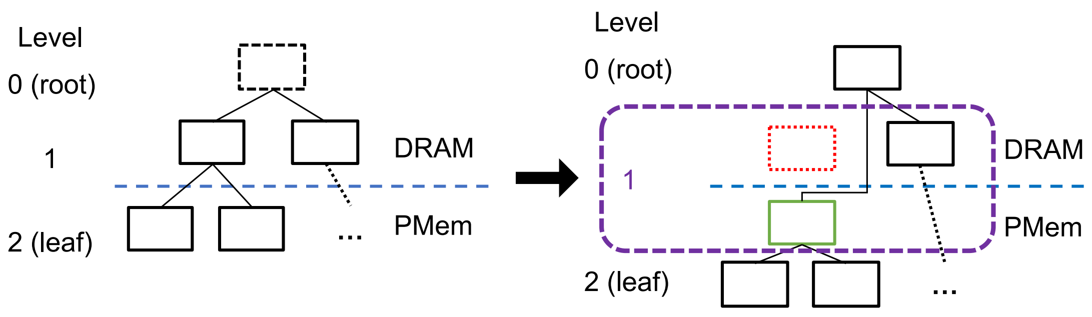


Figure 23: Root split exceeding the DRAM level limit with *level-bound adaptive persistence*.

process, the level has nodes in PMem and DRAM at the same time.

Since the DRAM levels are the upper levels, their maximum number of nodes is independent of the tree's height. This is beneficial because an upper bound for the memory consumption can be calculated with the *VNode* size. Table 3 lists the maximum number of nodes and upper bounds of DRAM consumption for configurations of a tree with $M = 38$ (with M = maximum node size). As the table shows, the upper bounds grow exponentially with each additional DRAM level. Between one and three maximum DRAM levels, the DRAM consumption only increases up to 1.8 MiB. While 1.8 MiB more or less DRAM consumption has no significant effect on an application, it demonstrates the exponential growth of the upper bound. From level three to four, the upper bound increases up to 68.8 MiB. From level four to five, it increases already to 2.6 GiB. Configuring limits in between is not possible. Additionally, the upper bound is only rarely reached in practice since it implies a filling degree of 100% for the most upper levels.

Max. DRAM levels	**Max. # volatile nodes**	**Uppper bound DRAM consumption**
1	1	1.3 KiB
2	39	48.8 KiB
3	1,483	1.8 MiB
4	56,355	68.8 MiB
5	2,141,491	2.6 GiB

Table 3: Upper bounds for *level-bound adaptive persistence* configurations.

5.1.2 Persisting Operation

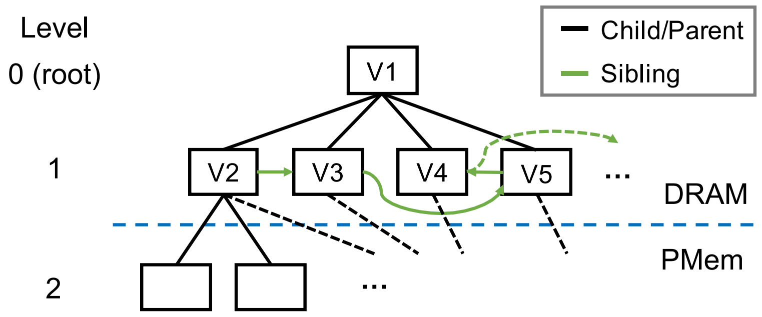


Figure 24: Example structure before level one is persisted.

A background thread does the copying of nodes from DRAM to PMem while other operations continue to work on the tree. As we describe in Section 4.1, the class of the node object determines if the object is persistent or not. Consequently, the volatile object in DRAM cannot just be copied, but a new persistent object has to be created. Multiple challenges arise for the background thread persisting the level. First, the background thread has to ensure that no node of the level still lies in DRAM when it finishes. Operations in between can cause a split in that level, resulting in new nodes for the level while the persisting thread is still running. Second, no operation should still work on a volatile node while the node is being copied and deleted. Third, persisting the level has to be a failure-consistent operation as well.

To have a defined order to traverse the nodes of a level, PeaR-Tree uses sibling pointers in nodes (compare Figure 14). Whenever a node splits, it sets its sibling pointer to the new node. Should the node already have a sibling pointer set, the new node receives the old sibling as its sibling pointer. Otherwise, the sibling pointer of the new node is *NULL*. Figure 24 depicts an example sibling pointer structure before the background thread starts persisting level one. As the nodes *V3*, *V4*, and *V5* show, the sibling pointer does not necessarily reflect the order of branches in the parent nodes. During a split, branches can be reordered. However, only the branches in the parent node are reordered, but not the sibling pointers in the child nodes. For each level, the *PeaRTree* object saves a pointer to the first node. Starting at that node, the background thread traverses the level and persists the nodes.

In the following, we prove that the linked sibling node list guarantees that the whole level is persisted when the persisting thread reaches the end. If no other operation modifies the list during the persisting process, the list is completely

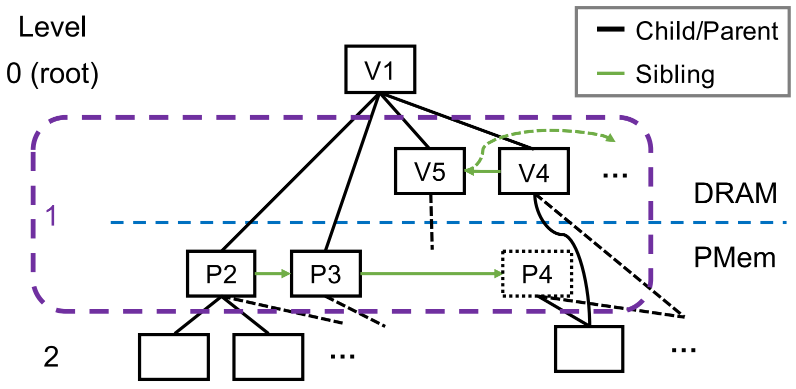


Figure 25: Example structure during the persisting process of *V4*.

traversed. Concurrently running insert operations can cause a split in a node of the level that is in the process of being persisted. The split adds a new node to the level and therefore modifies the linked sibling node list. We show that splits in our example on level one do not harm the completeness of the persisting process. Figure 25 displays the state of persisting node *V4*. A persistent copy, *P4*, was already created and linked with *P3*. However, the parent *V1* is still pointing to *V4*. Operations accessing the fourth branch in *V1* see the locked node *V4*. The persisting thread does not lock any other node. Since *V4* is locked, a split could only happen on the nodes, which are already persisted (*P2* and *P3*) or not yet persisted (*V5* and the following nodes).

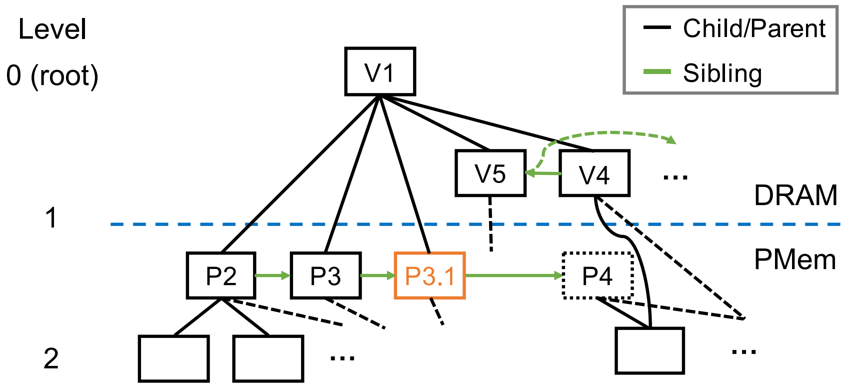


Figure 26: Persistent node split during the persisting process of *V4*.

If an operation causes a split of node *P3* the new node would be created as a persistent node since *P3* is persistent. Figure 26 shows the result of such a split. The linked sibling pointer list changed. *P3* points to *P3.1* and *P3.1* points to *P4*. It is guaranteed that *P3.1* points to *P4* because the sibling pointer of *P3* was set during its creation before it was unlocked and able to split. All of them are persisted and the persisting process of *V4* is not influenced. This proves that a split of a persisted node does not harm the completeness of an ongoing persisting operation at the same level.

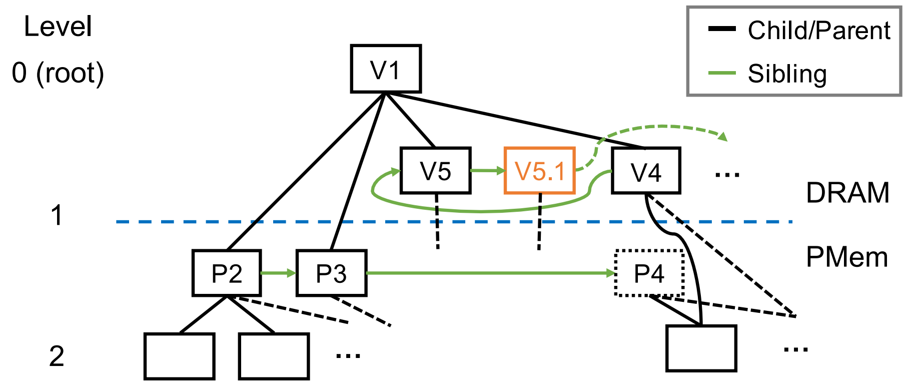


Figure 27: Volatile node split during the persisting process of *V4*.

If the not yet persisted node *V5* splits, a new volatile node *V5.1* gets created, as Figure 27 illustrates. Since it was created out of *V5*, the sibling pointer of *V5* now points to *V5.1*. Consequently, the sibling pointer of *V5.1* points to the old sibling of *V5*. *V4* does not change. When the persisting process is finished with *V4*, *V5* is pointing to *V5.1*. Therefore, both are persisted as well as the nodes after *V5.1*.

We show that the persisting process does not interfere with other operations by analyzing the pseudo-code of persisting a level in Listing 1. The listing contains only the part of the background persisting that traverses the nodes. Before the persisting of a volatile node starts, it is first marked as locked (Line 5). Other concurrent operations do not modify or access the node from that point on. In Line 6 all branches are copied to the new persistent object. If a split happens during the persisting process, the branch referencing the *VNode* object could be moved to a new parent node. To avoid data races when updating the branch in the parent node and setting the correct parent pointer, we lock the parent node. While we wait for the successful acquiring of the lock, we update the parent reference after each failed locking attempt (Lines 7-10).

Finally, the next volatile node that needs to be persisted is selected (Lines 13ff.). When the persisting thread reaches a *VNode* object that has no sibling, the level is completely persisted. The pointer to the first persistent node of the level and the number of the level are written to a persistent metadata file. After the information is persisted, the reference to the last persisted level is changed atomically to point to the persisted level. Since the change happens atomically, the persisting of the

Listing 1: Background Persister: Persisting all nodes of a level.

```
VNode* volatile_node = level_start_node;
PNode* new_persistent_node = PNode(locked=true);
bool has_sibling = true;
while(has_sibling) {
  volatile_node.lock = true;
  new_persistent_node.branches = volatile_node.branches;
  parent_node = volatile_node.parent_node;
  while(!parent_node.try_acquire_lock()) {
    parent_node = volatile_node.parent_node;
  }
  new_persistent_node.parent_node = parent_node;
  parent_node.lock = false;
  volatile_node = volatile_node.sibiling_node;
  has_sibling = (volatile_node != NULL);
  if(has_sibling) {
    new_persistent_node.sibiling_node = allocate_p_node();
  } else {
    new_persistent_node.sibiling_node = NULL;
  }
  new_persistent_node.lock = false;
  if(has_sibling) {
    new_persistent_node = new_persistent_node.sibiling_node;
  }
}
```

level happens atomically from the perspective of a recovery process. Therefore, the background persisting process is a failure-consistent.

5.2 Node-Bound Adaptive Persistence

Node-bound adaptive persistence describes a persistence strategy with a more fine-granular configurable persistence degree than *level-bound* persistence. It defines a maximum number of nodes that are placed in DRAM. The number can be set regardless of the node size and number of levels. As with the *level-bound adaptive persistence*, the upper part of the tree is always volatile, while the lower part is persistent. However, with *node-bound adaptive persistence*, levels can be persistent and volatile at the same time. In PeaR-Tree it is only one level at a time with a hybrid status, i.e., the hybrid level. Nodes themselves are either completely persistent or completely volatile. Therefore, the hybrid levels consist of a set of persistent nodes and another set of volatile nodes. All levels above the level are completely volatile and all levels beneath are persistent. Because this requires a different persisting concept than for *level-bound adaptive persistence*, we describe the new concept and operation in this section.

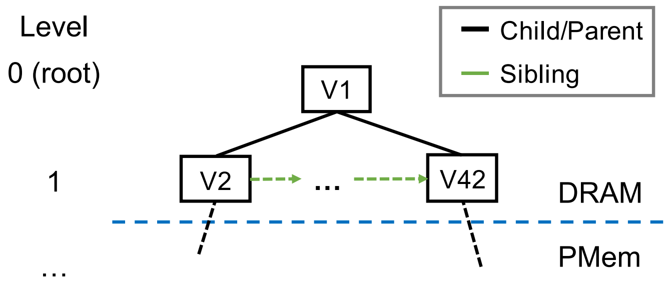


Figure 28: Example of a PeaR-Tree instance with *node-bound adaptive persistence*.

5.2.1 Persisting Concept

Initially, no nodes lie in DRAM since the first root node is a leaf node and lies in PMem. After the first split, the new root is placed in DRAM. All further splits place the new node in the same memory as the full node, no matter at which level. The *PeaRTree* instance counts the nodes that lie in DRAM. As long as the DRAM node count is not at the configured capacity, no nodes are moved. When the capacity is reached, nodes of the level above the leaf level are persisted. Figure 28 shows an example tree with $M = 46$ and a maximum DRAM node count of 42 (with M = maximum node size). *VNode* object *V42* is the last node that was created out of a split in DRAM. It is the 41st node in level one and, therefore, the 42nd node in DRAM. When it splits again and *V43* gets created, *V43* is placed

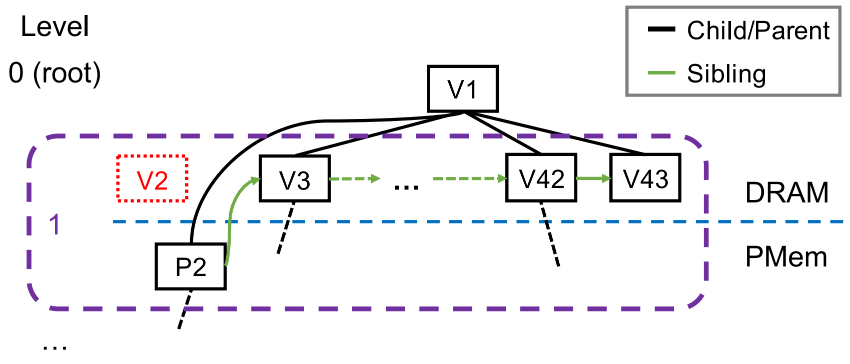


Figure 29: Example of persisting a node with *node-bound adaptive persistence*.

in DRAM because *V42* lies in DRAM. However, it is the 43rd node in DRAM, so another node needs to be moved to PMem. In Figure 29 we see that node *V2* gets persisted. It is the first node in the linked sibling node list of level one. The linked sibling node list determines the order in which nodes are persisted within a level. For each node that splits in DRAM, the next node in the linked sibling pointer node list gets persisted.

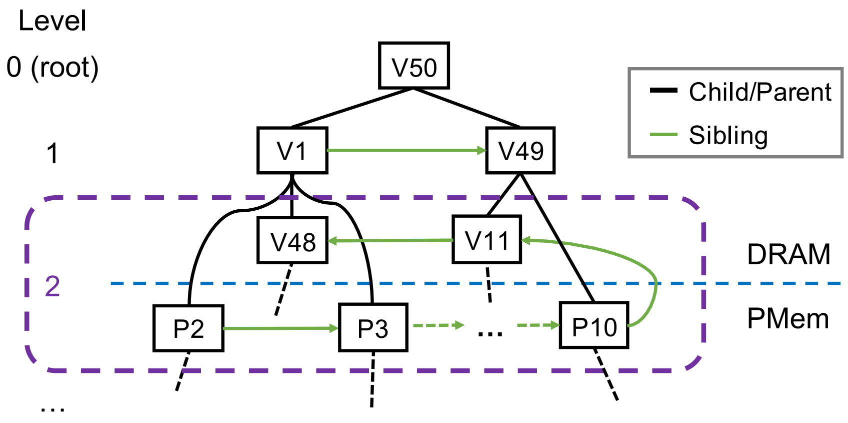


Figure 30: Example of a root split with *node-bound adaptive persistence*.

Figure 30 illustrates the situation of the example when the 46th split happens in level one, and the root node needs to split to fit all 47 nodes. The nodes of level one became the nodes of level two since the tree grew in height. For the additional nodes in DRAM, nine nodes of the old level one had to be persisted. Now level two is the hybrid level. The sibling pointer of the persistent node *P9* points to the volatile node *V10*, which is the next node to be persisted. This example demonstrates that the number of entirely volatile levels can vary during runtime.

Node-bound adaptive persistence allows PeaR-Tree to offer a fine-granular configuration of the DRAM consumption. *Level-bound adaptive persistence* DRAM limits can only be set with exponentially increasing step size depending on the maximum node size. The limit of *node-bound adaptive persistence* can be set with a precision of 1 MiB and guarantees to hold as many upper levels as possible in DRAM. If a level does not fit entirely into the DRAM budget, it becomes a hybrid level and as many nodes as possible are kept in DRAM. The only limit to configuring the consumption is the number of inner nodes. An instance configured with *node-bound adaptive persistence* can only hold as much nodes in DRAM as there are inner nodes. Leaf nodes are always placed in PMem to ensure the persistence of all inserted entries.

5.2.2 Persisting Operation

In a *node-bound adaptive persistent* configuration, the persisting process happens within the split operation. When a *VNode* object splits, the global nodes in DRAM counter is checked. If the counter is at capacity, the next node in the linked sibling node list gets persisted. The *Insert* operation causing the split calls the same persisting function as we present with Listing 1. Instead of looping through the linked sibling node list, only one node is persisted. We evaluate the options of persisting multiple nodes in a batch and using a background task in Section 6.3.4. Because both options do not show a better performance, we do not integrate them in PeaR-Tree. Afterwards, the last persisted node points to the volatile node that is persisted next. A reference to the last persisted node is stored in a global state so that the next *Insert* operation causing a split of a volatile node can persist the next node. If the last persisted node is the last node in the linked sibling node list of a level, the level above is persisted next. As a result, the *node-bound* persisting also persists the levels alongside the linked sibling node list, only in multiple steps. This process comes with the following challenges. First, it is essential for the process to persist a level completely before persisting nodes of another level. Second, the persisting process should not conflict with any other operation or cause a deadlock. Third, the process should not interfere with the failure consistency of a split and be failure-consistent itself.

As Section 5.1 shows, persisting a level along the linked sibling node list guarantees to persist all nodes of the level. Although the level is not persisted as one continuous operation but as an independent subprocess of a split, the assumption is still true. Analogous to the background persisting thread, a persist subprocess of a split locks a *VNode* object and waits for all operations to finish before it starts persisting it. With time in between the persisting subprocesses, a split of the last persisted node could happen before the next persisting subprocess starts. This causes the sibling pointer of the last persisted node to point to a different node object than the next *VNode* object that needs to be persisted. For that reason, the persisting subprocess traverses all sibling pointers until one points to a *VNode* object. Consequently, all volatile nodes of a level are persisted when the persisting subprocess finds a NULL pointer as the next sibling pointer.

The persisting process as part of the split is implemented similarly to the persisting process as a background process. Therefore, it guarantees that it does not come into conflict with other running operations. However, the persisting subprocess of a split could try to persist a *VNode* object, which is locked as part of the *Insert* operation that started it. To avoid a deadlock, the persisting subprocess breaks if the to-be-persisted node is currently locked due to a split. The following persisting subprocess then persists two nodes at once to restore the maximum amount of DRAM nodes.

As we show in Section 5.1, the process of persisting a whole level is failure consistent. The only difference for the persisting subprocess is that it happens within the insert operation. However, the persisting subprocess does not persist any of the splitting nodes and, thus, has no influence on the failure-consistency of the insert operation causing the split. If a failure happens during the persisting subprocess, the recovery process would recognize the only half persisted node. Therefore, the persisting subprocess is also failure consistent if it happens during an insert operation.

5.3 Recovering Adaptively Persisted Trees

When a PeaR-Tree instance is only partially persisted, it cannot be instantly recovered. Opposed to a fully persistent PeaR-Tree instance, the root node and its children are not persisted in PMem. They need to be reconstructed before the instance can accept queries again. To reconstruct the non-persistent parts after a restart, the information from the lower persistent nodes are used. Since PeaR-Tree guarantees persistence for a subtree if the root of the subtree is a persistent node, these subtrees can be treated as fully persistent subtrees during recovery. For the recovery process, the challenge arises to use the persistent subtrees as efficiently as possible to recover the entire tree structure. In the following sections, we first present an efficient recovery process for a PeaR-Tree instance with *level-bound adaptive persistence*. Then, we describe how this process needs to be modified for a PeaR-Tree instance with *node-bound adaptive persistence*.

5.3.1 Level-Bound Recovery

For a *level-bound* persisted PeaR-Tree instance, we know that the first persisted level is a completely persisted level. If a failure happens while the background thread is running, the level is not marked as persisted.

A persistent metadata file holds the pointer to the first node of the linked sibling list of the most upper persisted level. In Figure 31 we see an example of a tree with three volatile levels (zero, one, and two). The highest persistent level is level three. As depicted in pink, each persistent node still holds a pointer to the non-existent parent nodes, the nodes *V2* and *V3*. Even though the parent node objects do not

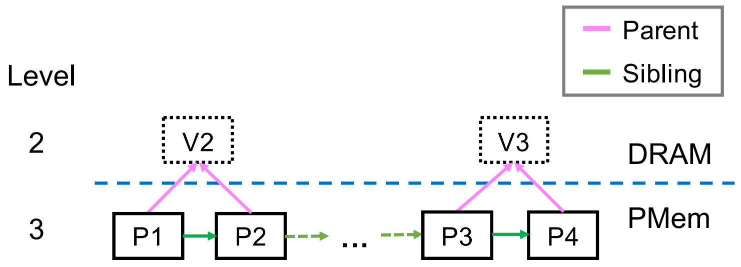


Figure 31: Persistent part of a PeaR-Tree instance persisted at level three.

exist, the information can be used to identify, which node of level three belonged to which node on level two.

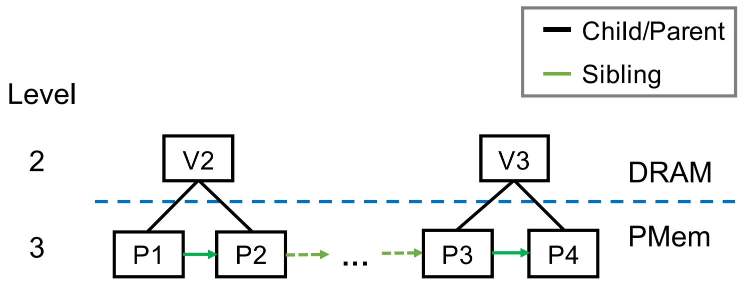


Figure 32: Restoring the parent nodes of the persisted part.

The recovery process starts traversing the linked sibling node list. It assigns each persistent node to a parent pointer in a map. Afterwards, for each parent pointer entry, a new volatile node is created and all persistent nodes belonging to the parent are added as branches, as shown in Figure 32. The nodes *V2* and *V3* are now existing objects again holding references to the nodes *P1*, *P2*, and the following nodes. To assign the correct MBR to each branch, the entire MBR array of the persistent node and the bitmap have to be read by the recovery process. The MBR of the persistent node can only be calculated with this information. After the linked sibling list of the highest persisted level is traversed, the level above is recreated. We further call the level the *recreated level*. Then, a new *PeaRTree* object is created with the root node starting at level one. The starting level of the root node for a recovered tree is the *recreated level* number plus one. In the left half of Figure 33 we see how the recreated node *V1* is inserted as a child to the root node *V1* of the newly created instance. Node by node, all other nodes of the *recreated level* are inserted to the new instance. The insert operation is the same routine as for a normal insert. However, it treats the level above the *recreated level* as the leaf level and the pointers to the *recreated level* as the inserted values. As a result, the tree structure above the *recreated level* is incrementally rebuilt.

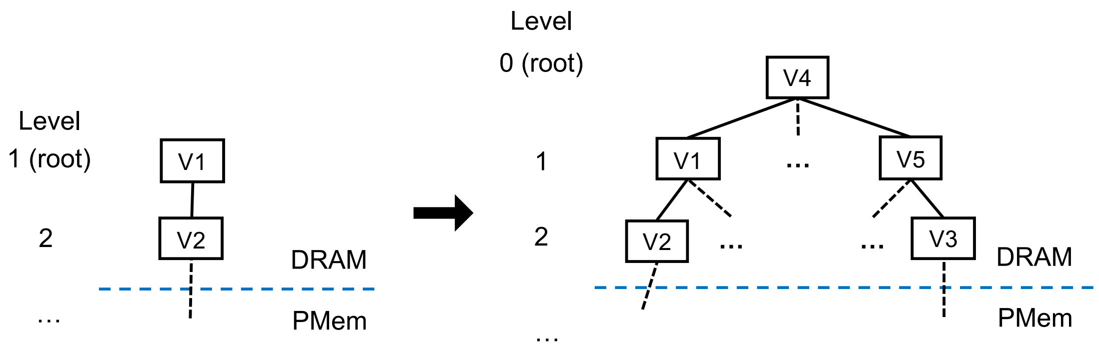


Figure 33: Rebuilding the tree structure by reinserting the recreated nodes.

The second half of Figure 33 illustrates the result. Since the initial root node *V1* split at a certain point, a new root *V4* was created. It has the designated root level number of zero. All recreated parent nodes of the last persisted level are part of the structure now. With their references to the persistent subtrees, the tree structure is recovered completely. While it is not guaranteed that the inner nodes are linked in the same structure as before the restart, all persistent subtrees are part of the structure and all MBRs are correct. Therefore, all values can be found and the tree structure is correct.

Last persisted Level	**Max. # Nodes Last Persisted Level**	**Recreated & Reinserted Nodes**	**Accessed PMem**
3	1,483	38	915 KiB
4	56,355	1,483	34 MiB
5	2,141,491	56,355	1.3 GiB
6	81,376,659	2,141,491	47.9 GiB

Table 4: Worst-case scenario costs for a recovery at a given level for $M = 38$.

The costs of this recovery process depend on the number of nodes in the last persisted level. Even if the last persisted level is not the leaf level, the size of the level depends on the number of inserted entries. However, an upper bound can easily be calculated as the maximum size of the level depends on the maximum node size. Table 4 shows the calculation broken down to the individual steps for M=38. From each node of the last persisted level, the MBR array has to be read from PMem in addition to the bitmap and the sibling and parent pointer. At most, these are M^l nodes (with l = number of last persisted level, and M = maximum node size). The upper bounds for the different last persisted levels are given in column *Max # nodes last persisted level*. For each parent pointer, a new node has to be allocated and initialized with branches and MBRs. In the worst case, M^{l-1} nodes are allocated, calculated in column *Recreated & Reinserted nodes*. Then these need to be inserted into a new tree structure one after another. The cost of inserting M^{l-1} entries into a PeaR-Tree instance are added to the worst-case scenario costs. The amount of data accessed on PMem is calculated as the following. For each node in the last persisted level, the whole MBR array is loaded. Each MBR is encoded with four floating-point values with a size 4 B each. In total, the array has a size of 608 B. Then the bitmap and the pointer pair are accessed, each having a size of 8 B. So for each node, the recovery process accesses 608 B + $3 * 8$ B = 632 B in the worst case. Again, we see that the upper bound costs increase exponentially with each additional level in DRAM, as with the upper bound DRAM consumption. In Section 6.3.3 we conduct an empirical analysis of the actual recovery costs.

5.3.2 Node-Bound Recovery

If a *node-bound* persisted PeaR-Tree instance needs to be recovered, the recovery process can rely on the same principles as for a *level-bound* recovery. Also, with *node-bound* persistence, each subtree is guaranteed to be persistent if the subtree's root is a persistent node. The only difference is that the recovery process cannot simply follow the linked sibling node list of the last completed level and recreate the parent nodes. Some of these parent nodes might be persistent nodes, making the effort of recreating them redundant. Figure 34 shows an example of a *node-bound* persisted PeaR-Tree instance before recovery. Level four is persisted completely, and level three is a hybrid level with *P4* and *P5* as persistent node examples. However, the parent *V2* on level three of *P1* and *P2* is only referenced but does not exist as a persistent object.

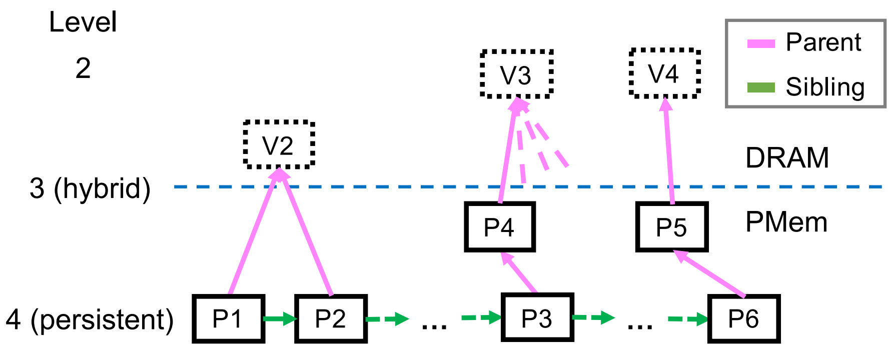


Figure 34: Persisted part of an instance with *node-bound adaptive persistence*.

The recovery process starts traversing the last entirely persisted level, level four, and build a map for parent nodes that need to be recreated, the *complete level parent map*. Since the address ranges for persistent objects are known, the recovery process can identify persistent objects based on the pointer address. If a node on level four has a persisted parent, e.g., the nodes *P3* and *P6*, their parents are added to a different map, the *hybrid level parent map*. The *hybrid level parent map* tracks parent and child nodes for the level above the hybrid level, level two in the example. Since not all children of a node on level two have to be persisted, the recovery process can find fewer than m persisted children on level three for a node on level two (with M = maximum node size, m = minimal node size, and $2 \leq m \leq M/2$). In Figure 34 this is visualized by the additional parent pointers pointing to *V3* while *V4* only has *P5* referencing it. We assume that in this example *V3* has more than m persisted children pointing to it. For all nodes on level two, where this is the case, the recovery process recreates the parent node. All other referenced nodes on level two are ignored and their persistent children are treated as recreated parents on level three for the remaining recovery process. The non-persistent children a node on level two might have had before the restart are not relevant. They are parents of persistent nodes on level four and are recreated through their reference, as we explain in the following.

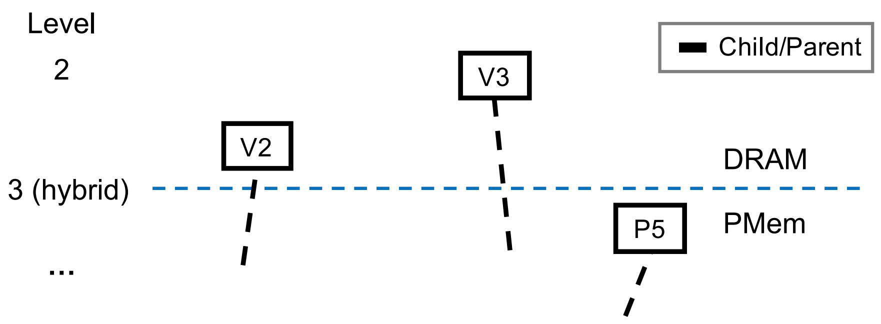


Figure 35: Reconstructing all parents having enough persistent children.

Figure 35 shows the subtree structure after the recovery process recreated parent nodes for all nodes, which had enough persisted children referencing them. *V2* now references the persistent nodes on level four, e.g., *P1* and *P2*. *V3* on level two references the persistent nodes on level three, e.g., *P4*. *P5* on level three with references to the persistent nodes on level four was not modified. As opposed to the *level-bound* recovery, the volatile part of the tree is rebuilt in three phases. In the first phase, parent nodes created based on the *hybrid level parent map* are inserted into a new instance. In the second phase, persistent nodes on the hybrid level with no recreated parents are inserted into the new instance. Finally, recreated parent nodes from the *complete level parent map* are inserted. As Figure 36 shows, the new instance is initialized with a root having the level number of the last complete persisted level plus two. In our example, node *V3* is the only node recreated from the *hybrid level parent map*, as we see on the left side. In this phase, all nodes are added with an insert routine that treats the nodes two levels above the hybrid level as the leaf nodes. In the next two phases, the level above the hybrid level is treated as the leaf level. Thereby, the persistent nodes of the hybrid level and the recreated nodes from the *complete level parent map* are inserted on their initial level. The second part of Figure 36 gives an example of how the tree could look after phase three. As with the *level-bound* recovery, it is not guaranteed that the inner nodes have the same structure as before the restart. Especially persistent nodes on the hybrid level, for which the parent was not recreated, can be inserted into another node on the level above. In the example *P5* is now a child of *V3*. However, this does not influence the correctness of the tree. All MBRs leading to *P5* were recalculated and the subtree under *P5* has not changed. All values in the subtree *P5* are still covered by the MBRs of their parents and can therefore be found.

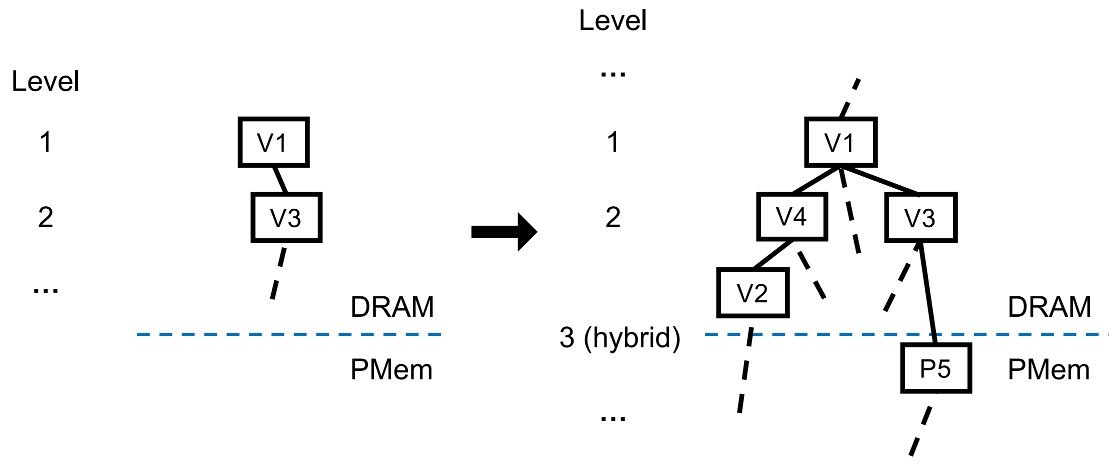


Figure 36: Rebuilding the tree structure by reinsert the persisted and recreated nodes.

The costs of the *node-bound* recovery depend on the tree's original structure. A formal worst-case cost analysis is not possible solely based on the number of not persisted nodes. Therefore, we conduct a qualitative analysis of the cost factors. Regardless of the structure, every node of the level beneath the hybrid level has to be accessed. The recovery process traverses this level and accesses the parent and sibling pointer of each node. Opposed to the *level-bound* recovery not every MBR array of the level is scanned. For the persistent nodes in this level with a persistent parent, the MBR does not need to be calculated. Although less data needs to be accessed for the level as with the *level-bound* recovery, the number of random reads increases still exponentially with each additional level in DRAM. For each persistent node in the hybrid level, the parent pointer and the MBR array are accessed. With the MBRs the MBR in the recreated parent for the reinsertion is calculated. The more persistent nodes in the hybrid level share the same parent in the level above, the fewer nodes from the hybrid level need to be reinserted. In Section 40 we extend this analysis with empirical measurements.

5.4 Summary

So far, we have presented related work about persistent one-dimensional index structures designed for the use on PMem [12, 13, 36, 37, 40, 55]. They prove that persistent index structures on PMem can achieve nearly the same access performance as volatile in-memory indexes. Additionally, they offer very fast or instant recovery after a restart. NV-Tree introduces the concept of selective persistence [55]. Selective persistence stores only the parts of the index structure persistently that are crucial to recovering the entire structure. While this increases the recovery time compared to a fully persistent structure, it significantly increases runtime performance.

With PeaR-Tree, we extend concepts of FPR-Tree to have a failure-consistent and concurrent R-Tree with full functionality [13]. The PeaR-Tree design allows light-weight failure-consistency. For each operation, a sequence of recovery states is defined. Each of the states has a recovery procedure in case a node in the particular state is detected after a restart. With the recovery states, all operations do not need copy-on-write or log mechanisms to modify a node. Since the state detection happens at runtime, no recovery process is needed during start-up for a fully persistent PeaR-Tree instance.

Further, we add the concept of storing parts of the tree structure in DRAM to improve the runtime performance, similar to the concept of selective persistence [55]. In contrast to selective persistence, we allow limiting the DRAM consumption regardless of the number of contained entries. For that, we introduce *adaptive persistence*. *Adaptive persistence* is a concept that dynamically adapts nodes' storage placement to the available DRAM and the size of the tree structure. We introduce *level-* and *node-bound adaptive persistence*. They either limit how many of the upper levels lie in DRAM or how many nodes. If new nodes are created in DRAM, PeaR-Tree follows the linked sibling list of a level to move nodes of a level to PMem. As a result, all levels beneath a persistent node are persistent as well. This allows to efficiently recover *adaptive persistent* PeaR-Tree instances by rebuilding the tree structure with the persisted subtrees. With *node-bound adaptive persistence* the DRAM consumption can be set with a precision of 1 MiB. The larger the DRAM limit, the more nodes leverage the performance of DRAM compare to PMem. However, the more nodes lie in DRAM, the more nodes need to be recovered after a restart, increasing the recovery time.

6 Evaluation

In this section, we evaluate the concept of *adaptive persistence*. Therefore, we measure the details of the PeaR-Tree design. Our measurements follow the same methodology, which we describe in Section 6.1 together with the technical details of our test setup. Section 6.3 presents the results of our benchmarks in detail. First, we discuss the comparison of PeaR-Tree to other systems. Second, we analyze *adaptive persistence* towards its influence on the runtime performance and recovery time. Third, we assess the design decisions for PeaR-Tree by comparing different options against each other.

6.1 Methodology

In all our measurements, we use the NYC Yellow Cab taxi data set [42]. It contains start- and endpoints for all Taxi Rides conducted in New York City. Because it is a real-world dataset with billions of data points, it is used by many others to evaluate spatial data processing [13, 33, 47]. We limit the datasets to 200 million entries from the year 2009. The limited dataset includes start- and endpoints, and we do not differentiate between the two in the following evaluation. All points are encoded as single-precision floating-point values, i.e., 4 B.

For our evaluation we consider the operations *Insert* and *Query* like other R-tree research work does [3, 4, 8, 9, 22]. We measure the throughput in a multi-user setting for *Insert* operations and *Query* operations with a point selection predicate. For *Query* operations with a range selection predicate, we measure the latency for different selectivities in a multi-user setting.

During the setup of each benchmark, a PeaR-Tree instance is created and filled with the values of the dataset. After the setup is completed, the operation to be measured is executed with a single thread or multiple threads, depending on the configuration. This procedure is repeated five times for each benchmark, and the measurements are taken as the mean of all runs.

If not described otherwise, the PeaR-Tree instance for a benchmark is configured as follows. The maximum node size is set to 38 and the filling degree to 30%. For all benchmarks with a DRAM limit larger than 0 MiB, a *node-bound adaptive persistent* configuration with in-place persisting and a persisting batch size of ten is used. The selection and details of these configurations are explained in Section 6.3.4 and Section 6.3.5.

As a comparison to state-of-the-art persistent R-tree systems, we use eFIND, FAST-Rtree, and FPR-Tree [8, 13, 50]. Since the implementations for eFIND and FAST-Rtree are only available as GiST indexes for Postgres, we use the FESTIval extension for Postgres. It is an extension to benchmark different index implementations within Postgres. The measurements happen within an SQL function that measures the library function call to the index operation. All library functions are implemented in C. While eFIND and FAST-Rtree systems are designed to be used with SSDs, we measure them with PMem as a persistent storage medium. Therefore, they are more comparable to PeaR-Tree and FPR-Tree. While we use the available implementations for eFIND and FAST-Rtree, FPR-Tree does not have a publicly available implementation. We implement the FPR-Tree as close as possible to the paper. In some details, we deviate from their description or make assumptions to guarantee correctness, as we describe in the following.

In our FPR-Tree implementation, we store the 8 B metadata as an atomic. Otherwise, data races occur during concurrent execution. Further, we add an atomic flag indicating if a thread currently holds the nodes mutex. *Query* operations do not need to acquire the node mutex since they do not modify the node. To avoid that a *Query* operation reads a node that is concurrently modified by another operation, we use the atomic flag. Before we recursively call the *Insert* function on the child of an inner node, we capture the metadata of the child. If the metadata changed after the mutex of the child node is acquired, it means the MBR of the node has changed. If the update happened after the branch of the node in the parent was selected, the selection is incorrect. To avoid an incorrect structure, we start over the path selection at the root node. Changes to MBRs along the path do not harm the correctness of the tree because they only increased the area. Moreover, should the selection have already been made based on the correct MBR the same branch is selected again. If the original branch is not found in the parent node during a split, the *Insert* restarts at the root as well. In this case, another concurrent *Insert* operation splits the parent and moved the original branch to a new node. After all updates to the branch array, we use *std::atomic_thread_fence(std::memory_order_release)* [14, 15]. And before each access, we use *std::atomic_thread_fence(std::memory_order_acquire)*. The fences guarantee that changes to the branch array made by concurrent operations are visible to the accessing operation.

6.2 Setup

Our benchmarking system has two Intel(R) Xeon(R) Gold 6240L CPU @ 2.60GHz CPU sockets. However, we only use one of the two to avoid NUMA-related influences on our measurements. One socket has 18 physical and 36 virtual cores. Each physical core has a 64 KiB L1 and a 1 MiB L2 cache. All cores of one socket share a 24.75 MiB L3 cache. As DRAM, 84.4 GB DDR4 is available for each socket. Additionally, each socket has six Intel Optane DC Persistent Memory 256 GB DIMMs 100 Series connected in interleaved mode.

The server runs Ubuntu 20.04.2 LTS. All executions are pinned via *numactl* to the physical cores of one socket and to the respective memory region. Only the benchmarks evaluating hyperthreading are pinned to all 36 cores of a socket. The code is written in C++ with the C++20 standard enabled. For compilation, we use *gcc* 9.3.0 with the *cmake* 3.16.3 release build configuration, i.e. -O3.

6.3 Microbenchmarks

PeaR-Tree allows setting different configurations, reflecting our design decisions. With microbenchmarks, we compare the influence of these configurations on the operation performance isolated from other influences. In this section, we present the results of these microbenchmarks. Therefore, we compare the different PeaR-Tree configurations against each other as well as against other systems. For each benchmark, we measure three types of operations. These are:

***Insert* operation microbenchmark**
Multiple worker threads create an instance with 100 million entries during the setup. In the measured benchmark, 100 million additional entries are inserted in single *Insert* operations. If not specified otherwise, all benchmarks are executed with 18 threads in parallel. This is the number of physical cores on our CPU. Section 6.3.1 shows that hyperthreading can achieve improvements in the throughput. However, we do not use it for the measurements comparing specific aspects, as it adds an additional external factor to the measurements. For multiple running threads, each thread inserts the same amount of entries such that in total 100 million entries are inserted.

Point *Query* microbenchmark
All 200 million entries are inserted in single-threaded execution mode in the same order for every benchmark. Thereby, the same tree structure is created for all

benchmarks. During the measured benchmark, 500 000 points from the inserted entries are queried. If not specified otherwise, the queries are executed with 18 threads in parallel. Each thread queries 500 000 points.

6.3.1 Existing Systems

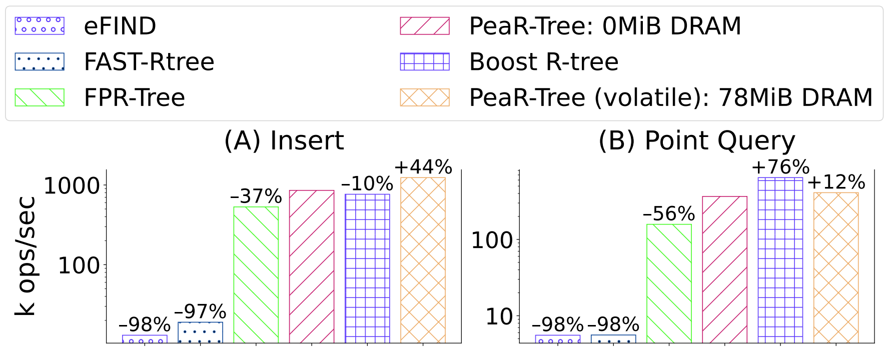


Figure 37: Single-threaded runtime performance comparison of PeaR-Tree to other systems.

With this first measurement, we compare the single-threaded operation performance of PeaR-Tree with state-of-the-art R-tree implementations. As a comparison, we use eFIND [8], FAST-Rtree [50], FPR-Tree [13], and Boost R-tree [18]. eFIND and FAST-Rtree are R-tree implementations optimized for the usage of flash-based storage like SSDs. FPR-Tree is the only other failure-consistent R-tree optimized for the usage on PMem, to the best of our knowledge. Boost R-tree is a single-threaded volatile R-tree implemented in C++. While it does not use any persistence mechanisms, it provides a good comparison to assess the pure R-tree operation performance.

Since FESTIval does not support the measurement of concurrent operations, we only compare the single-threaded performance of eFIND and FAST-Rtree. Although eFIND and FAST-Rtree run on PMem as well, they are orders of magnitude slower than all other systems. Therefore, we only take a small sample of the actual test data set of 1.5 million entries to limit the benchmarking time. Additionally, FESTIval does not support defining a setup routine for the benchmark. For that reason, we measure the insertion to an empty instance as opposed to the described procedure in Section 6.1 of prefilling the instance.

Figure 37 shows the throughput for the *Insert* and point *Query* operation. The y axis is on a log scale and the percentages mark the relative difference to the

PeaR-Tree: 0MiB DRAM configuration. *0MiB DRAM* describes the configured DRAM maximum consumption. For both operations, eFIND and FAST-Rtree are orders of magnitude slower than the comparison systems. In contrast to FPR-Tree, PeaR-Tree, and Boost R-tree, these systems are designed to write to storage media with block accesses and a high latency like SSD. If the underlying storage medium has a lower latency, the accesses themselves become faster. However, the systems still employ buffers and logs and write in blocks. This overhead accounts for most of the performance difference.

PeaR-Tree: 0MiB DRAM outperforms FPR-Tree by nearly 2x for *Insert* throughput. The difference comes from a more efficient *ChooseSubtree* algorithm of PeaR-Tree, which takes up most of the computation time. FPR-Tree's memory layout is not optimized to reduce accessed data in PMem. The pointer and MBR are stored next to each other in memory. During the *ChooseSubtree* algorithm, the MBRs of a node are accessed altogether without other branch information. PeaR-Tree stores them separately to reduce the overhead of unused data during node access.

Although Boost R-tree is not a persistent R-tree index, we include it as a state-of-the-art volatile R-tree. To draw an even comparison, we configured *PeaR-Tree (volatile): 78MiB DRAM* that also the leaf level is kept in DRAM, making it a completely volatile structure. The comparison shows that the structure of PeaR-Tree's operations does not impose an overhead for the *Insert* throughput compared to state-of-the-art implementations. As Figure 37 (A) shows, even the *Insert* throughput of *PeaR-Tree: 0MiB DRAM* is higher than the throughput of the pure DRAM Boost R-tree. On the other side, Boost R-tree outperforms *PeaR-Tree (volatile): 78MiB DRAM* for *Query* throughput. The different behavior for the two operations is explained by the different *ChooseSubtree* algorithms. Boost R-tree uses the R*-Tree *ChooseSubtree* algorithm during the *Insert* operation. On the leaf level, it calculates the least overlap enlargement. While the resulting MBRs have a lower overlap, which favors *Query* operations, calculating the overlap has a quadratic time complexity. PeaR-Tree uses the original R-tree *ChooseSubtree* algorithm during insert. For all levels, it selects the subtree in a node where the MBR needs the least area enlargement. Calculating the area enlargement for all branches of a node has only a linear time complexity. However, the MBR overlap on the leaf level is not as small as with the R*-Tree algorithm.

The results of the *Query* benchmarks exhibit the same behavior as the *Insert* benchmark results for the other systems. eFIND and FAST-Rtree show a performance orders of magnitude slower than the other systems. PeaR-Tree achieves over 50% higher throughput than FPR-Tree. The volatile version of PeaR-Tree has a higher throughput. But the throughput is not as good as the one from Boost

R-tree.

The comparison to other state-of-the-art persistent R-tree systems shows that PeaR-Tree outperforms them in all operations. Since their performance is orders of magnitude slower, we conclude that the optimizations to an R-tree employed for SSDs cannot be transferred to PMem. Therefore, we exclude these systems from all further measurements. The *Insert* and *Query* operations perform even comparably to the Boost R-tree. If PeaR-Tree runs completely volatile, it even achieves a 60% higher *Insert* throughput than Boost R-tree. This proves that the additional concepts of PeaR-Tree do not produce an overhead for the R-tree algorithm routines.

Scalability

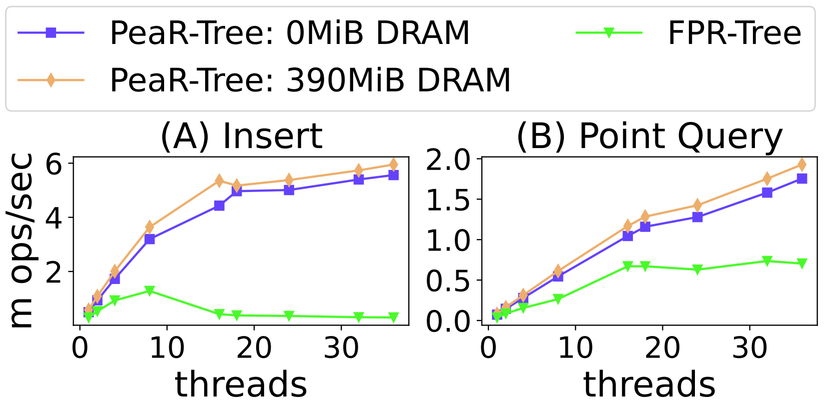


Figure 38: Scalability of PeaR-Tree and FPR-Tree for the *Insert* and *Query* operations.

To assess the scalability of PeaR-Tree, we compare two PeaR-Tree configurations with FPR-Tree. We use one configuration of PeaR-Tree where all inner nodes lie in DRAM and the leaf nodes in PMem. This configuration guarantees the persistence of all inserted entries while placing as many nodes in DRAM as possible. *PeaR-Tree: 390MiB DRAM* has enough DRAM capacity so that all inner levels of an instance containing the 200 million entries fit into DRAM. Since FPR-Tree is a fully persistent R-tree, we add the PeaR-Tree configuration *PeaR-Tree: 0MiB DRAM* to this evaluation as well. For all configurations, we conduct the microbenchmarks for the *Insert* and *Query* operations.

Figure 38 (A) shows the results for the throughput measurements. We see that FPR-Tree does not scale linearly for *Insert* as opposed to PeaR-Tree. After reaching a peak of 1.28 million inserts per second at eight threads, the *Insert* throughput of FPR-Tree declines again. Further, the increase is not as linear as with the PeaR-Tree configurations. Since the upper levels are comparably small, we expect to already have contention during branch updates or node splits for small thread counts. As a concurrency control mechanism, FPR-Tree uses mutexes. With more threads running, the probability increases that a mutex is locked while another thread tries to lock it. When this collision happens, expensive system calls are necessary to suspend the thread and wake it up again when the mutex is available again. After the peak, the overhead introduced by the interruptions is higher than the additional computation time, which is added with each thread. Therefore, we see the throughput declining even below the single-threaded throughput.

PeaR-Tree uses a more lightweight concurrency control. We use atomic variables in combination with busy waiting as a concurrency control mechanism. As the results show, this introduces less overhead than mutexes. Up to a thread count of eight threads, both configurations nearly scale linearly. The *PeaR-Tree: 390MiB DRAM* configuration achieves a 10-15% higher throughput for all thread counts than the *PeaR-Tree: 0MiB DRAM* configuration. Here, we see the lower DRAM accesses latency advantage since the inner nodes all lie in DRAM. After that, the throughput increases further but not as linear as before. With each additional thread, it is more likely that a thread has to wait for another thread to unlock a node or branch. After 18 threads, the virtual cores of the CPU are utilized with hyperthreading. This drastically increases the pressure on the cache as threads start to share the first and second-level cache. The increased contention causes more cache misses and more stall time, slowing down the throughput increase. The throughput only increases by 12% from 18 to 36 threads for both configurations.

For the *Query* operation, PeaR-Tree outperforms FPR-Tree with 2-3 times more throughput for point queries. FPR-Tree scales well up to a thread count of 16 threads but stagnates afterwards. PeaR-Tree scales even linearly up to a thread count of 18 and can achieve a 50% throughput increase between 18 and 36 threads. PeaR-Trees node layout favors comparisons of all MBRs of a node at the same time. With PeaR-Tree the MBRs are stored in a separate array from the other branch information. FPR-Tree stores the information of a branch together. If all MBRs are compared, not only the MBRs but also the child pointer are loaded from PMem. More loaded data results in more accessed PMem cache lines. Also, the CPU cache is filled with more unused data resulting in earlier cache evictions. The pressure on the cache explains why the FPR-Tree *Query* throughput plateaus after 16 threads, while PeaR-Tree increases with hyperthreading. Again, we observe that the *Query* throughput of the *PeaR-Tree: 390MiB DRAM* configuration is about 10% higher due to the better read latency of DRAM.

In conclusion, we see that PeaR-Tree achieves a two to five times higher throughput for the *Query* and *Insert* operation than FPR-Tree. Important factors are the more lightweight concurrency control and cache-optimized node layout and operation structure. The cache-optimized node layout allows PeaR-Tree to scale its throughput with hyperthreading. FPR-Tree's *Query* throughput stagnates in the hyperthreading thread count range and the *Insert* throughput even decreases. For both operations, the *PeaR-Tree: 390MiB DRAM* configuration performs up to 10% better.

6.3.2 Adaptive Persistence Degree

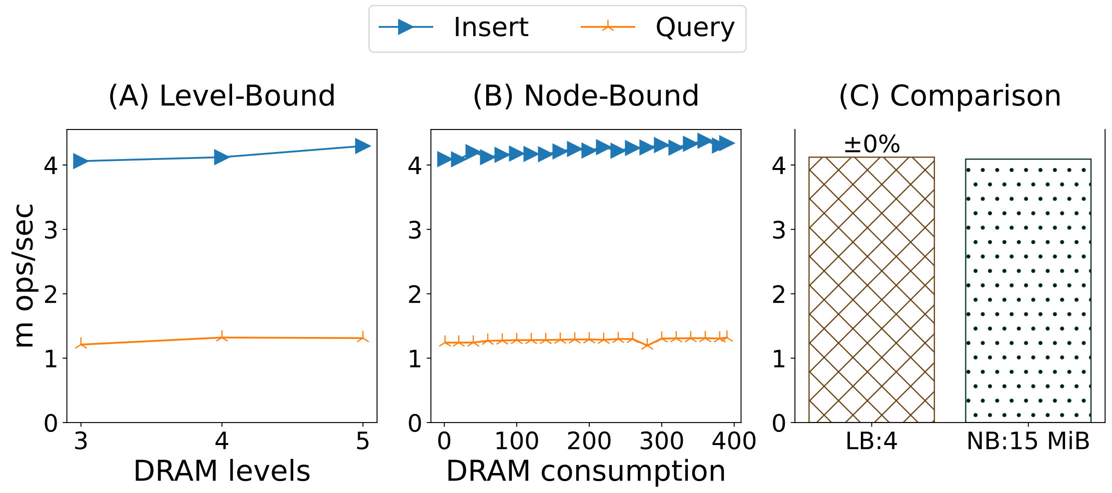


Figure 39: Persistence degree evaluation for different *adaptive persistence* configurations.

Adaptive persistence allows setting a static limit for the DRAM consumption. In this section, we investigate the influence of different limits on the performance of the *Insert* operation. We execute the benchmarks in this section with a thread count of nine. Higher thread counts show anomalies for both operations. As we do not know the source of the anomalies, we cannot use the results to draw conclusions about the persistence degree influence. For *level-bound adaptive persistence*, we evaluate the DRAM limits of three to five DRAM levels. Three DRAM levels are the smallest limit PeaR-Tree allows. Theoretically, one or two DRAM levels are possible. A two or one DRAM level limit results in at most $M+1$ nodes in DRAM at the same time (with M = maximum node size). The persisting process will interfere with a root node split. PeaR-Tree does not implement the additional concurrency control mechanisms to ensure serializability of all constellations of concurrent operations in this case. The tree structure has six levels, including the leaf level. Consequently, at most five levels lie in DRAM. As the DRAM limit for *node-bound adaptive persistence*, we evaluate the range from 1 MiB, the minimal setting, to 390 MiB. 390 MiB equals the size of all inner nodes of a PeaR-Tree instance filled with the test data. The total size of a PeaR-Tree instance for the given dataset is 9.76 GiB. It is independent of the configured *adaptive persistence* degree since persistent and volatile nodes have the same size of 1280 B. Consequently, the amount of PMem consumption ranges from 9.76 GiB for a DRAM consumption limit of 1 MiB to 9.38 GiB for a DRAM consumption limit of 390 MiB. Larger DRAM limits show no effect with our test data since no more nodes are suitable to be placed in DRAM. In this range, we measure the performance for different limits with a step size of 20 MiB.

Figure 39 (A) shows the *Insert* and *Query* throughput for *level-bound adaptive persistence* increases steadily with more levels in DRAM. With a DRAM level limit of five, the *Insert* throughput reaches a 7.5% higher throughput than the *Insert* throughput of a three DRAM levels limit. The *Query* throughput for five DRAM levels lies about 8% above the throughput of three DRAM levels. The actual DRAM consumption for a DRAM level limit of three and four lies below 10 MiB for this dataset. For a DRAM level limit of five, the consumption is 390 MiB. However, this depends on the number of inserted entries. As we explain in Section 5.1.1, PeaR-Tree guarantees for a DRAM level limit of five that the DRAM consumption will be below 2.5 GiB. For a limit of three DRAM levels, the guaranteed upper bound is 66.9 MiB. This means if we increase the upper bound with *level-bound adaptive persistence* by a factor of 38, the performance increase is only 7-8% for both operations.

Node-bound adaptive persistence allows setting a DRAM limit that is more fine grained. From Figure 39 (B), we see that the *Insert* throughput with a DRAM limit of 390 MiB is 6% higher than with a limit of 1 MiB. The *Query* throughput shows the same difference. A DRAM limit of 1 MiB equals 0.01% of the total data size. For a DRAM limit of 390 MiB, about 4% of the instance is stored in DRAM. With the increase from 0.01% of the data in DRAM to 4%, the throughput increased by 6%. The DRAM limits in between exhibit the same relation. This shows that the *adaptive persistence* DRAM limit leverages the available DRAM space regardless of the relation between the overall memory consumption and the DRAM limit size. The only limit is the number of nodes that are above the leaf level.

In Figure 39 (C) we compare the *node-bound adaptive persistence* strategy and *level-bound adaptive persistence* strategy regarding their *Insert* throughput. We exclude the *Query* throughput of this measurement, as the persistence strategy only comes to play during the *Insert* operation. Their configuration ensures that both consume the same amount of DRAM. We choose a DRAM level limit of four and the respective DRAM consumption limit of 15 MiB for the *node-bound* configuration. A DRAM level limit of three has a DRAM consumption of less than 1 MiB. With a DRAM level limit of five, no nodes are persisted during the benchmark run. As the results demonstrate, both *adaptive persistence* strategies have the same costs of persisting for an equal DRAM limit.

The study of different persistence degrees proves that adaptive persistence allows controlling the DRAM consumption and operation performance accordingly. Both *adaptive persistence* strategies achieve the same performance improvement with more nodes in DRAM. Therefore, the overhead of persisting does not depend on the persisting strategy. However, the relative throughput improvement of *level-bound adaptive persistence* is not proportional to the increase of the upper bound for the memory consumption. With *node-bound adaptive persistence* the throughput increases proportionally to the DRAM consumption limit.

6.3.3 Recovery

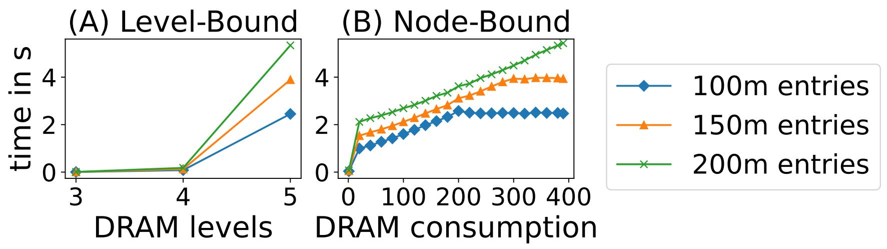


Figure 40: Recovery time of *partial persisted* PeaR-Tree instances.

In an *adaptive persistence* configured instance of PeaR-Tree, not all nodes are persisted to PMem. After a restart, this part of the tree structure needs to be reconstructed. In this section, we compare different DRAM limits and tree sizes with their respective recovery time. We measure the recovery time of an instance with 100 million, 150 million, and 200 million entries. PeaR-Tree only implements a single-threaded recovery process, so all recovery times are measured in single-threaded execution. As the configured limits, we use the same limits as we use for the evaluation in Section 6.3.2. For the *node-bound* configuration, we use DRAM limits between 1 MiB and 390 MiB and for the *level-bound* configuration, we use a limit of three, four, and five DRAM levels. The measurements include the pure rebuilding time of a previously created PeaR-Tree instance. They do not include any system setup time as the system is not restarted during the benchmark.

In Figure 40 the recovery times for the *node-bound* configurations and *level-bound* configurations are shown. The recovery time of an instance with a DRAM level limit of three or four lies in the milliseconds' range regardless of the number of contained entries. This is orders of magnitudes faster than the recovery times for a DRAM limit of five DRAM levels. Due to the tree structure, the number of nodes that need to be recovered increases exponentially per level. With three DRAM levels to recover, between 250 and 450 nodes need to be recovered. With four DRAM levels, there are already 6,000 to 12,000 nodes to recover. And with a DRAM limit of five levels, the number rises to 160,000 to 320,000 nodes to recover. This explains the exponential increase between the levels in the recovery time.

For the *node-bound adaptive persistence* configurations, we observe the same effect. With an increasing number of nodes in DRAM, the recovery time increases. However, the different sizes of the datasets reveal another factor towards the recovery time. The amount of nodes to be recovered is the same for a given DRAM limit, regardless of the contained entries. Nevertheless, the larger the instance, the longer the recovery is. With *node-bound adaptive persistence* levels are persisted partially, allowing to utilize the available DRAM as efficiently as possible. During recovery, the persisted nodes of the hybrid level do not need to be recovered and even provide information to reconstruct nodes on the level above. However, the last completely persisted level still needs to be read entirely to recover the remaining nodes of the hybrid level, as we explain in Section 5.3. Therefore, the recovery time depends on the size of the last completely persisted level for *node-bound* recovery as well. The more entries are inserted into an instance, the larger this level is. In this benchmark, the last persisted level for all configurations with a DRAM limit higher than 1 MiB is the leaf level. The levels above only sum up to a size between 6 and 10 MiB. This also explains the large difference between the recovery time for a DRAM limit of 1 MiB and 20 MiB. For a tree with a limit of 1 MiB, the last completely persisted level is the level above the leaf level. The difference in size is that for a limit of 1 MiB, 160,000 to 320,000 persistent nodes have to be read and for a limit of 20 MiB, four to eight million persistent nodes.

The *node-bound* configurations need the same time to recover as the *level-bound* configurations for the respective DRAM consumption. As with the operation performance, the choice of the *adaptive persistence* strategy does not influence the recovery performance. Additionally, the recovery of hybrid levels improves the recovery time in relation to the number of persistent nodes in the hybrid level.

The measurements in Figure 41 illustrate the importance of the parent pointer structure. We measure the recovery time for a *PeaR-Tree: 390MiB DRAM* configured instance, using parent pointers and without using them. A recovery without parent pointers takes 72% longer than a recovery with using parent pointers. During runtime, each node maintains a reference to its current parent node. While the reference becomes invalid after a restart for volatile nodes, the information is used to recover the node structure. The nodes of the last persisted level are assigned to recreated volatile nodes according to their parent pointers. Without the information, which nodes of the last persisted level had the same parent, a new tree needs to be bottom-up by inserting all nodes of the persisted level to a new instance. With the information of the parent pointers, only the recreated nodes need to be reinserted into the new instance. Due to the properties of a tree structure, this reduces the number of reinserted nodes exponentially. Still, for both variants, the complete last level is loaded from PMem. As we explain for the measurement of Figure 40 (B), the loading time makes up a large portion of the recovery time.

The recovery measurements demonstrate that *adaptive persistence* influences the recovery time of a given instance proportionally to the configured DRAM consumption limit. Different amounts of inserted values influence the recovery time as well, given a fixed DRAM consumption limit. For one thing, the recovery time depends on the number of volatile nodes before a restart. With a higher DRAM consumption limit, this number increases. For another, the recovery process has to load the last completed level from memory. Even if this level has the same level number in instances with different sizes, the number of nodes is higher for instances with more entries regardless of how many levels the instance has.

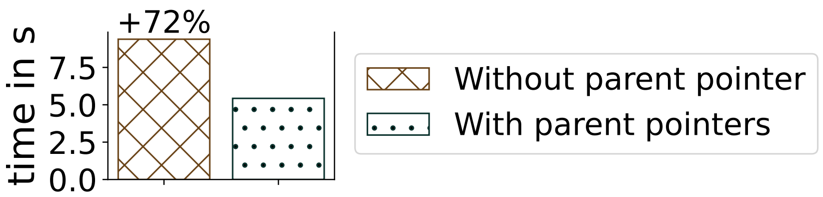


Figure 41: Comparison of a recovery with and without using the parent pointer information.

6.3.4 Persisting Volatile Nodes

With this evaluation, we asses which persisting strategy is the best for *node-bound adaptive persistence*. Apart from the in-place persisting, which we introduce

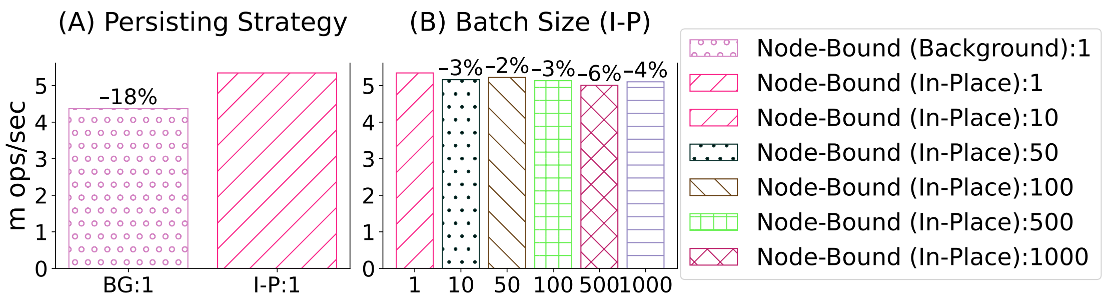


Figure 42: Comparison of different persisting strategies for *node-bound adaptive persistence*.

in Section 5.2.1, a background persisting strategy is an option. Instead of the thread that caused the DRAM limit's overflow, a new thread persists the next node. Additionally, we test different persisting batch sizes. A batched persisting process only starts when enough nodes over the DRAM limit are in DRAM that a complete batch can be persisted at once. *Level-bound* adaptive persistence does not have these different possibilities for the persisting process because it only starts when a whole level needs to be persisted. Therefore, we exclude it from this evaluation. For *node-bound* adaptive persistence, we use the *PeaR-Tree: 280MiB DRAM* configuration. With 280 MiB DRAM capacity, the DRAM is fully utilized after the preparation of inserting 100 million entries. During the measured part of the benchmark, each inner node split leads to a node being persisted.

First, we compare the *Insert* performance of background persisting against in-place persisting. Figure 42 (A) shows that the background strategy (BG:1) only achieves a 18% lower throughput than the in-place persisting strategy (I-P:1). With background persisting, additional overhead is introduced by dispatching a new thread. Further, an additional thread increases the possibility of lock contentions with other *Insert* operations.

As the label indicates, the comparison of background persisting and in-place persisting is executed with a batch size of one node. Figure 42 (B) shows that one is the optimal batch size, out of the compared sizes for in-place *node-bound* persisting. The results show that larger batch sizes for persisting do not increase the overall throughput. During the persisting process, the split node stays locked and other threads cannot access it. Larger batch sizes increase the time for a persisting process. The longer the split node stays locked, the more likely it is that other threads try to access it and are blocked by the lock. We exclude the results for background persisting in this chart, as they all have a lower *Insert* throughput. Although the throughput increases with larger batch sizes as the *Insert* operation continues after dispatching the thread, it still is lower than any batch size with the in-place persisting.

Overall, we conclude that the persisting process introduces less overhead if it is done in place. Persisting in batches does not improve the *Insert* throughput for the in-place persisting. On the opposite, larger batch sizes increase the time, the *Insert* operation conducting the persisting process locks the splitting node.

6.3.5 Persistent Node Performance

In this section, we assess the design decisions in PeaR-Tree for the persistent node layout. To measure the influence on the persistent nodes the best, we execute all evaluations in this section with a fully persistent configuration of PeaR-Tree. First, we compare different node sizes against each other and analyze their factor towards the runtime performance. Second, for the best node size, we identify the best filling degree given our test dataset. Third, we evaluate the influence of cache aligned nodes and MBRs.

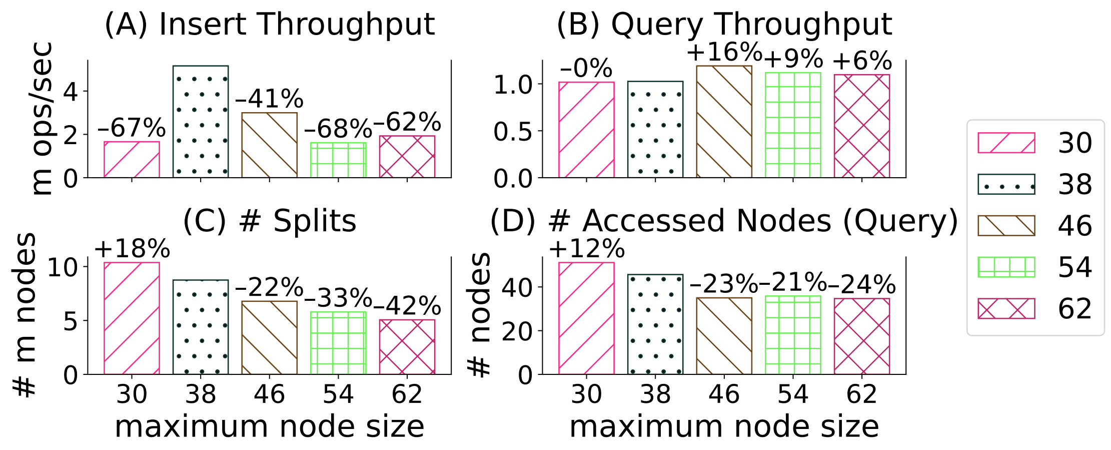


Figure 43: Evaluation of the maximum node size for PeaR-Tree.

Node size
Figure 43 (A) visualizes the results of the runtime performance for different maximum node sizes. All persistent node objects in PeaR-Tree are aligned to 256 B, which is the PMem cache line size. The node size determines how much space the branch and MBR array take up. Because the MBRs of a node are often accessed all at once, we design the node memory layout so that the MBR array is aligned to 256 B as well. To have a minimal amount of padding per node, we evaluate

the following node sizes: 30, 38, 46, 54, 62. Persistent nodes with these sizes only need 16 B of padding. Other node sizes have more padding per node if the node and the MBR array is aligned to 256 B. The node sizes 22 and 14, which also only need 16 B of padding, are excluded. They show a lower throughput for both operations.

The filling degree for all node sizes is 30% since this is the optimal filling degree given the test dataset. Above each throughput bar, the relative difference to the throughput of the maximum node size 38 is shown. Because 38 is the set maximum node size for all other benchmarks. The second-best maximum node size of 46 has a 41% lower *Insert* throughput. However, the *Query* throughput for a maximum node size of 46 is 16% higher than for the size of 46. The results for the other node sizes do not show a clear trend. The *Insert* throughput for larger node sizes than 38 declines at first but then inclines from the maximum node size of 54 to 62. For the *Query* throughput, larger *Query* sizes achieve a higher throughput, but after a node size of 46, the *Query* throughput declines again.

The non-linear distribution of the results indicates that the maximum node size influences the runtime performance in multiple ways. First, the maximum node size determines the size of the MBR array in memory. Although the maximum node size is not always fully utilized, the larger the node is, the more often the whole array has to be loaded from memory. For each *Insert* operation, as many MBR arrays as there are levels in the tree are loaded from PMem. The *ChooseSubtree* routine selects on each level a branch until it reaches the leaf level. In the case of a split or concurrency conflict, this number increases. This favors smaller nodes since one *Insert* operation needs to load less data from memory than with a larger maximum node size. On the other side, the smaller the maximum node size, the more often nodes need to split. As Figure 43 (C) shows, the smaller the maximum node size, the more splits happened during the creation. In conclusion, these two mechanisms have contradicting trends for the best node size. The same two mechanisms influence the results of the *Query* operation. As Figure 43 (D) shows, a large maximum node size with fewer nodes leads to fewer accessed nodes during a *Query* operation. On the other side, nodes with more children have larger MBRs, which leads to a higher overlap. For that reason, we do not see that the number of accessed nodes steadily decreases but stabilizes around 35. Apart from the pure number of accessed nodes, the size of the nodes influences the *Query* operation as well. Therefore, we see a throughput decrease for maximum node sizes larger than 46.

Despite the lower *Query* throughput, we choose 38 as the maximum node size for all other benchmarks. The *Insert* throughput increase compared to a size of 46 is higher than the *Query* throughput decline.

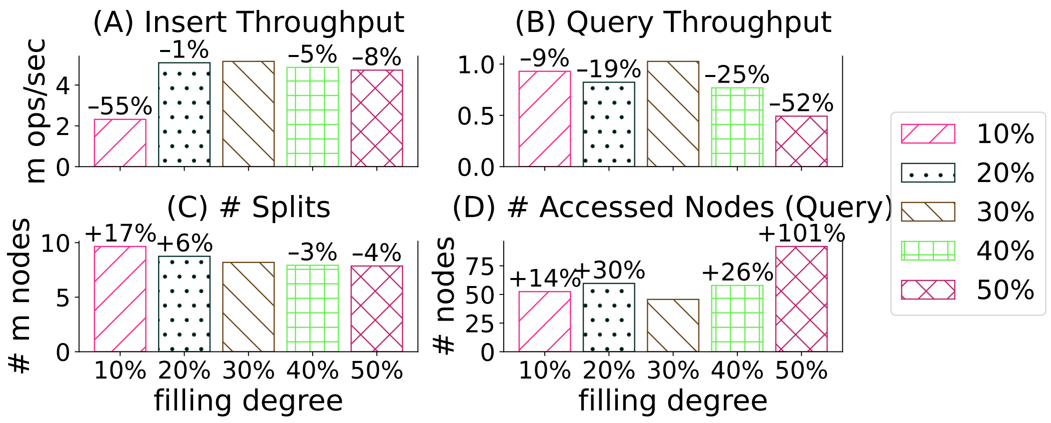


Figure 44: Evaluation of the filling degree for a maximum node size of 38.

Filling Degree

The minimal filling degree of a node is determined by the ratio of m and M (with M = maximum node size, m = minimal node size, and $2 \leq m \leq M/2$). This allows setting a minimal filling degree of up to 50%. The minimal filling degree determines when a node merges if entries are deleted. It also sets the limits for the split algorithm of how it can distribute the branches among the splitting and the new node. After a split, both nodes must have at least m branches. A filling degree of 50% defines that the split algorithm needs to distribute all branches evenly, as both nodes need to be filled at least 50% after the split. With a smaller filling degree, the split algorithm has the option to distribute the branches more unevenly. The R*-tree split algorithm, which PeaR-Tree uses, does that if the resulting MBRs have a smaller total area. The distribution after a split defines when the node will split next. With an even distribution, both nodes are likely to split at the same time again. While an uneven distribution increases the chances that the more filled node splits sooner. We evaluate five filling degrees from 10% to 50%. All these degrees are evaluated with a maximum node size of 38 because it is the optimal node size for our dataset. For each filling degree, we measure the *Insert* and *Query* throughput, as well as the number of splits for the *Insert* benchmark and the number of accessed nodes for the *Query* benchmark.

Figure 44 (A) shows that a filling degree of 30% achieves a slightly better *Insert* performance. Except for a filling degree of 10%, all other filling degrees achieve an *Insert* throughput that is only a few percent lower than the throughput for 30%. A filling degree of 10% has a 55% percent lower throughput. Figure 44 (C) shows that with a filling degree of 10%, 17% more splits occur as for the other filling degrees. The smaller filling degree produces splits, where one node holds very few branches and the other many branches. For one thing, more splits require more write operations to PMem. For another, each split potentially blocks other concurrent running operations.

From Figure 44 (B) we see that a filling degree of 30% achieves the best *Query* throughput as well. The comparison to Figure 44 (D) shows that the filling degree influences how the tree is structured. With a small filling degree, more options to split a node are possible that would not be possible with a higher filling degree. Nevertheless, the R*-tree split algorithm, used by PeaR-Tree is not an exhaustive algorithm comparing all possible split distributions. It uses metrics to find a split with low overlap and area at the same time. In its selection process, margins and areas of multiple distribution possibilities are compared in groups. With more options to assess, the compared groups grow, and other options of the group possibly outweigh an optimal option. For a filling degree of 30%, the tree has a structure with less overlap than with other filling degrees, and the *Query* operations need to access fewer nodes. This translates into a higher *Query* throughput.

For both operations, a filling degree of 30% achieves the highest throughput. Therefore, we set this filling degree for all other benchmarks. Still, for other data, the same mechanisms apply. Smaller filling degrees allow more possibilities to split a node, which can lead to a better structure.

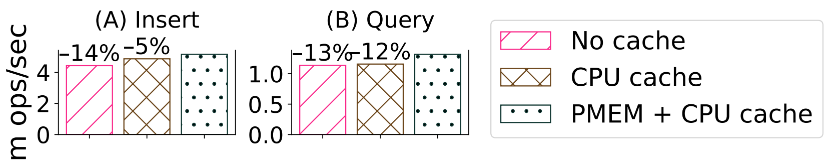


Figure 45: Comparison of different alignments of the persistent node in PeaR-Tree.

Cache Alignment

We evaluate an alignment of nodes and the contained MBRs to different caches. The CPU cache has a size of 64 B and the PMem has a size of 256 B. Consequently, if the objects are aligned to the PMem cache, they are also aligned to the CPU cache. The alignment to the PMem cache represents our node design, which we describe in Section 4.1.2. To measure the unaligned versions, we shift the pointer address for each object by an offset. As we allocate the memory in large chunks, we can add an offset to the first address of the chunk without any overhead. For the *No cache* aligned version, we set the offset to 24 B. With the offset, the nodes and MBRs are not aligned to 64 B nor 256 B. The offset for the *CPU cache* version is 128 B. Because 128 B is a multiple of 64 B, the nodes and MBRs stay aligned to the CPU cache but are not aligned to the PMem cache.

As Figure 45 shows, the configuration aligned to both caches achieves the highest throughput for *Insert* as well as for *Query*. For the *Insert* operation, the alignment to the PMem cache only has a small effect with an improvement of 5%. However, if the nodes are not aligned to both caches, the *Insert* throughput drops by 14%. On the other hand, for the *Query* throughput, if the node is not aligned to the PMem cache, the throughput decreases already by 12%. That is about the same when the node is not aligned to any cache. This shows that both alignments influence the overall runtime performance. For each MBR during a *Query* operation only four floating point values are compared. During an *Insert* operation multiplications, substractions, and comparisons are computed for each MBR. Therefore, the *Insert* operation can better rearrange the load instructions for the next PMem cache line. Hence, we see a larger decrease in *Query* throughput than in *Insert* if the nodes and MBRs are not aligned to the PMem cache.

6.4 Discussion

In this section we extensively evaluate PeaR-Tree. Our measurements prove that placing part of the tree structure in DRAM improves the *Insert* throughput over a tree structure entirely stored in PMem up to 10% percent. The only limit is the number of inner nodes since leaf nodes are not stored in DRAM by PeaR-Tree. However, the *Query* throughput is not affected by the different storage placement. The lower access latency for PMem is not the limiting factor for the *Query* operation. On the other hand, with an increased DRAM consumption, the recovery time increases proportionally. Throughout the benchmarks, we study the performance of the two different presented *adaptive persistence* strategies, *level-bound* and *node-bound adaptive persistence*. Both achieve the same results for different DRAM consumption. For the actually configured DRAM limit, *node-bound adaptive persistence* has a far better ratio of increased DRAM consumption limit to increased *Insert* throughput. We identify optimal settings for the configuration of the persisting batch size, maximum node size, and filling degree. The evaluation of these parameters also stresses the importance of an architecture optimized for PMem. If the nodes are not aligned to the respective cache sizes, the performance gain from nodes placed in DRAM is negated. Our conclusion is that *adaptive persistence* improves the *Insert* performance of an already optimized R-tree structure while controlling the DRAM consumption at the same time. The range of configurable limits is small compared to the overall memory size of an instance. Nevertheless, in systems where multiple instances of an index are kept in memory together with other applications, DRAM space becomes a critical resource requiring to be controlled mindfully.

7 Conclusion

In this thesis, we present PeaR-Tree, a persistent, failure-consistent, and concurrent R-Tree with support for *adaptive persistence*. We introduce *adaptive persistence* as a new concept to limit the DRAM consumption when storing the tree index structure distributed over PMem and DRAM. A static limit for the DRAM consumption is defined and PeaR-Tree dynamically adapts the storage placement of nodes as the tree structure grows. The nodes placed in DRAM leverage the better access performance of DRAM. At the same time, the memory consumption of the more costly DRAM resource is limited.

First, we present the details of the PMem technology and how it differs from other memory and storage technologies. Then, we motivate the need for spatial indexes in comparison to one-dimensional indexes. We describe one of the most prominent spatial indexes in detail, the R-Tree, and discuss some of its variants. Based on the R-Tree concept, we describe the design of PeaR-Tree as a persistent, failure-consistent, and concurrent spatial index. Finally, we introduce the concept of *adaptive persistence*. It has two strategies for limiting DRAM consumption. With one strategy, the number of levels in DRAM can be limited, level-bound *adaptive persistence*. With the other strategy, the number of nodes in DRAM can be limited regardless of the levels, node-bound *adaptive persistence*. We examine the concepts with respect to their implementation and implications for limiting DRAM consumption and the recovery process. While level-bound *adaptive persistence* uses a more simple recovery process, the DRAM consumption limit is only an upper bound and hardly ever fully utilized. The limit defined in levels only allows coarse-grained steps and the level of a tree needs to be fully utilized so that the limit is reached. On the other hand, node-bound *adaptive persistence* allows fine-grained limits in steps of 1 MiB. If enough nodes are in the tree, it is also fully utilized.

Our evaluations show that PeaR-Tree outperforms state-of-the-art persistent R-Trees optimized for flash storage by orders of magnitudes. In contrast to PeaR-Tree, these systems do not assume that the persistent storage is byte-addressable and, therefore, deploy additional mechanisms, like logs and buffers. Compared to FPR-Tree, a persistent R-Tree optimized for PMem, PeaR-Tree achieves up to 20 times more throughput for *Insert* and 3-4 times more throughput for the *Query* operation. Different to FPR-Tree, PeaR-Tree uses a node layout optimized for PMem access characteristics and a more lightweight concurrency control. With *adaptive persistence*, PeaR-Tree can leverage the better access performance for parts of the tree structure that lie in DRAM. Further evaluations show that the runtime performance and recovery time behave proportionally to the DRAM limit.

The higher the DRAM limit is, the more runtime performance PeaR-Tree achieves. But on the other side, the recovery takes longer after a restart. Thereby it does not make a difference if the DRAM limit is set through a node or level limitation. This shows that node-bound *adaptive persistence* offers more flexibility than level-bound *adaptive persistence* without introducing any overhead during runtime. Still, with node-bound *adaptive persistence* the complexity of the implementation increases.

With both *adaptive persistence* strategies, the arbitrary distribution of nodes between the inner levels and the leaf level is the limiting factor. As it is typical for a tree index structure, nearly all nodes are on the leaf level, which needs to be in PMem to guarantee the persistence of all entries. A subject for further research is how to guarantee the persistence of all entries so that nodes of the leaf level can be placed in DRAM as well. We introduced *adaptive persistence* as a concept for R-Trees. The extension to other tree index structures is possible with only little modifications. It is an interesting topic for future work if the concept is generalizable apart from tree index structure to data structures of all kinds.

References

[1] AnandTech. Intel launches optane dimms up to 512gb, 2018. `https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here` [Accessed: 2021-04-30].

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. volume 11, pages 553–565, 2018.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, page 322–331, New York, NY, USA, 1990.

[4] Norbert Beckmann and Bernhard Seeger. A revised r*-tree in comparison with related index structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 799–812, New York, NY, USA, 2009.

[5] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment*, 14(9):1544–1556, 2021.

[6] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[7] bit tech. Samsung 32gb solid state drive. `https://bit-tech.net/reviews/tech/storage/samsung_32gb_solid_state_drive/2/` [Accessed: 2021-05-04].

[8] Anderson C Carniel, Ricardo R Ciferri, and Cristina DA Ciferri. A generic and efficient framework for flash-aware spatial indexing. *Information Systems*, 82:102–120, 2019.

[9] Anderson C Carniel, Ricardo R Ciferri, and Cristina DA Ciferri. Festival: A versatile framework for conducting experimental evaluations of spatial indices. *MethodsX*, 7:100695, 2020.

[10] SAP HANA Central. Hana memory usage, 2021. `http://www.hanaexam.com/p/hana-memory-usage.html` [Accessed: 2021-04-30].

[11] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[12] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, 13(12):2634–2648, 2020.

[13] Soojeong Cho, Wonbae Kim, Sehyeon Oh, Changdae Kim, Kwangwon Koh, and Beomseok Nam. Failure-atomic byte-addressable r-tree for persistent memory. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):601–614, 2020.

[14] cppreference. std::atomic_thread_fence - cppreference.com. https://en.cppreference.com/w/cpp/atomic/atomic_thread_fence [Accessed: 2021-06-19].

[15] cppreference. std::memory_order - cppreference.com. https://en.cppreference.com/w/cpp/atomic/memory_order [Accessed: 2021-06-19].

[16] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing persistent memory bandwidth utilization for olap workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China*, Virtual Event, China, 2021.

[17] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[18] Barend Gehrels, Bruno Lalande, Mateusz Loskot, Adam Wulkiewicz, and Oracle and/or its affiliates. boost::geometry::index::rtree, 2017. https://www.boost.org/doc/libs/1_65_1/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost__geometry__index__rtree.html [Accessed: 2021-04-30].

[19] The PostgreSQL Global Development Group. Postgresql 13.0 documentation: Chapter 64. gist indexes, 2020. https://www.postgresql.org/docs/13/gist-intro.html [Accessed: 2020-11-01].

[20] GSAP. Spatial data (sql anywhere) - sap help portal. https://help.sap.com/viewer/0e60f05842fd41078917822867220c78/1.0.11/en-US/c80d4c846e1b10148c4983790acef053.html [Accessed: 2021-05-03].

[21] Ralf Hartmut Güting. An introduction to spatial database systems. *the VLDB Journal*, 3(4):357–399, 1994.

[22] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, page 47–57, New York, NY, USA, 1984.

[23] Jim Handy. Intel's optane dimm price model, 2019. https://thememoryguy.com/intels-optane-dimm-price-model/ [Accessed: 2021-04-30].

[24] Joseph M. Hellerstein. *Generalized Search Tree*, pages 1222–1224. Springer US, Boston, MA, 2009.

[25] IBM. Ibm informix spatial datablade module user's guide - ibm dokumentation. https://www.ibm.com/docs/de/informix-servers/11.5?topic=SSGU8G_11.50.0/com.ibm.spatial.doc/overview1002477.htm [Accessed: 2021-05-03].

[26] IBM. R-tree index user's guide, 2018. https://www.ibm.com/support/knowledgecenter/en/SSGU8G_12.1.0/com.ibm.rtree.doc/rtree.htm [Accessed: 2020-11-01].

[27] IBM. R-link trees and concurrency, 2019. https://www.ibm.com/docs/en/informix-servers/14.10?topic=method-r-link-trees-concurrency/ [Accessed: 2021-06-23].

[28] Intel. Clwb — cache line write back. https://www.felixcloutier.com/x86/clwb

[Accessed: 2021-04-30].

[29] Intel. Sfence — store fence. `https://www.felixcloutier.com/x86/sfence` [Accessed: 2021-04-30].

[30] Peiquan Jin, Xike Xie, Na Wang, and Lihua Yue. Optimizing r-tree for flash memory. *Expert Systems with Applications*, 42(10):4676–4686, 2015.

[31] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proceedings of the second international conference on Information and knowledge management*, pages 490–499, 1993.

[32] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 500–509, San Francisco, CA, USA, 1994.

[33] Andreas Kipf, Harald Lang, VN Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A Boncz, Thomas Neumann, and Alfons Kemper. Adaptive main-memory indexing for high-performance point-polygon joins. 2020.

[34] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 546–557, 2002.

[35] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. Str: A simple and efficient algorithm for r-tree packing. In *Proceedings 13th International Conference on Data Engineering*, pages 497–506, 1997.

[36] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+ trees: optimizing persistent index performance on 3dxpoint memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, 2020.

[37] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020.

[38] Yanfei Lv, Jing Li, Bin Cui, and Xuexuan Chen. Log-compact r-tree: an efficient spatial index for ssd. In *International Conference on Database Systems for Advanced Applications*, pages 202–213, 2011.

[39] Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.

[40] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 31–44, 2019.

[41] Vincent Ng and Tiko Kameda. The r-link tree: A recoverable index structure for spatial data. In *Proceedings of the 5th International Conference on Database and Expert Systems Applications*, page 163–172, 1994.

[42] City of New York. Tlc trip record data - tlc. `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page` [Accessed: 2021-05-04].

[43] Oracle. Spatial and graph features in oracle database — oracle. `https://www.oracle.com/database/technologies/spatialandgraph.html` [Accessed: 2021-05-03].

[44] Oracle. Mysql 8.0 reference manual, 2020. `https://dev.mysql.com/doc/refman/8.0/en/` [Accessed: 2020-11-01].

[45] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386, 2016.

[46] PostGIS. Postgis — spatial and geographic objects for postgresql. `https://postgis.net/` [Accessed: 2021-05-03].

[47] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. Gpu-based parallel r-tree construction and querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 618–627, 2015.

[48] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, pages 17–31, 1985.

[49] SAP. Persistent memory - sap help portal. `https://help.sap.com/viewer/6b94445c94ae495c83a19646e7c3fd56/2.0.04/en-US/1f61b13e096d4ef98e62c676debf117e.html` [Accessed: 2021-04-30].

[50] Mohamed Sarwat, Mohamed F Mokbel, Xun Zhou, and Suman Nath. Fast: a generic framework for flash-aware spatial trees. In *International Symposium on Spatial and Temporal Databases*, pages 149–167, 2011.

[51] Chris Terman. L14: The memory hierarchy, 2015. `https://computationstructures.org/lectures/caches/caches.html` [Accessed: 2021-05-11].

[52] Les Tokar. Samsung 980 pro gen 4 nvme ssd review (1tb/250gb) - 7gb/s speed with cooler temps — the ssd review, 2020. `https://www.thessdreview.com/our-reviews/nvme/samsung-980-pro-gen-4-nvme-ssd-review-1tb-250gb-7gb-s-speed-with-cooler-temps/3/` [Accessed: 2021-05-04].

[53] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Building blocks for persistent memory. *The VLDB Journal*, 29:1223–1241, 2020.

[54] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 169–182, 2020.

[55] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level sys-

tems. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 167–181, 2015.

Erklärung (Declaration of Academic Honesty)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Die selbstständige und eigenständige Anfertigung versichert an Eides statt:

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used. The independent and unaided completion of this thesis is affirmed by affidavit:

Potsdam, 24.06.2021

Nils Hendrik Thamm