Vestein Dahl

Handling big spatial data

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2019

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Vestein Dahl

Handling big spatial data

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

In recent years, modern services and applications handle more and more multi-dimensional data than ever before. Satellites, mobile devices, social networks and in the future IoT devices, generate huge amounts of data tagged with time, date and location. Handling large volumes of multi-dimensional data creates new challenges for today's data management systems that tries to index, store and analyze such data efficiently in real-time.

Transforming random writes to sequential writes have been increasingly important for insert-intensive workloads. Traditional index structures such as B^+-trees and R-trees perform very poorly with large quantities of random writes because of their usage of in-place writes for updates and insertion. Modern database systems have therefore implemented the Log-Structured Merge-tree (LSM) [1] in their storage layer [2–4].

Managing multi-dimensional data has gained attention with the development of advanced database systems which require high real-time throughput and efficiency for processing tasks or transactions. Techniques to improve performance include efforts to reduce the dimensionality of the data, because the memory models of modern computers are one-dimensional [5].

This thesis implemented an experimental LSM-tree data structure which incorporates a R-tree index in the disk component. By utilizing the properties of space-filling curves such as the Hilbert curve and Z-Order curve, it is possible to map multi-dimensional geometric objects to one dimension and create a sequential order which conserves locality. A storage layer structure was suggested in the thesis, additionally two different merging strategies, threading and other minor considerations when tuning for write throughput.

Sammendrag

De siste årene har moderne tjenester og applikasjoner måtte håndtert mer og mer flerdimensjonale data enn noen gang tidligere. Satellitter, mobilenheter, sosiale nettverk og fremtidige IoT-enheter genererer store mengder data merket med tid, dato og sted. Håndtering av store mengder flerdimensjonal data skaper nye utfordringer for dagens databasesystemer som forsøker å indeksere, lagre og analysere slike data effektivt i sanntid.

Å gjøre om tilfeldig skriving til sekvensiell skriving har blitt stadig viktigere for innsetting av store mengder data. Tradisjonelle indeksstrukturer som B^+-trær og R-trær takler store mengder med tilfeldig skriving svært dårlig på grunn av "in-place" skriving for oppdateringer og innsetting. Moderne databasesystemer har derfor tatt i bruk Log-Structured Merge-tree (LSM) [1] i lagringslaget [2–4].

Behandling av flerdimensjonal data har fått mye oppmerksomhet den siste tiden på grunn av utviklingen av avanserte databasesystemer som krever høy gjennomstrømning og effektivitet for behandling av oppgaver eller transaksjoner i sanntid. Teknikker for å forbedre ytelsen inkluderer metoder for å redusere antall dimensjoner til dataen, fordi minnemodellene til moderne datamaskiner er endimensjonale [5].

Denne oppgaven implementerte en eksperimentell LSM-tre datastruktur som inkorporerer en R-tre indeks i diskkomponenten. Ved å benytte egenskapene til romfyllingskurver som Hilbert Curve og Z-Order-Curve, er det mulig å forandre flerdimensjonal geometriske objekter til én dimensjon og lage en sekvensiell ordning som beholder lokaliteten. En lagringslagstruktur ble foreslått i oppgaven, i tillegg til to forskjellige sammenslåingsstrategier, tråding og andre vurderinger å ta hensyn til for høy skriveytelse.

Preface

This Master's Thesis is written for the Department of Computer Science (IDI) at Norwegian University of Science and Technology. The research is conducted by Vestein Dahl, under the supervision of Professor Svein Erik Bratsberg.

I would like to thank Svein Erik Bratsberg for his helpful assistance, ideas, technical advice and feedback throughout the semester.

Finally, I would like to thank my family for always supporting me.

Vestein Dahl

Trondheim, June 2019

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, modern services and applications handle more and more multi-dimensional data than ever before. Satellites, mobile devices, social networks and in the future IoT devices, generate huge amounts of data tagged with time, date and location. Handling large volumes of multi-dimensional data creates new challenges for today's data management systems that tries to index, store and analyze such data efficiently in real-time.

Traditional relational databases have usually been using B-trees, which have been the de facto standard access method in all relational systems for years. While being great for indexing one-dimensional data and doing point-queries, it can not efficiently index spatial data and do range-queries.

In 1983, the original R-tree was described by Guttman [6] and expanded the concept of B+-trees to a structure which can handle dynamic organization of d-dimensional objects. It paved the way for spatial databases and geo-information systems to store and query multi-dimensional data [7]. But the R-tree and most of its variants uses in-place writes like B-trees, resulting in expensive random writes to disk during updates.

Transforming random writes to sequential writes have been increasingly important for insert-intensive workloads. Traditional index structures such as B+-trees perform very poorly with large quantities of random writes because of their usage of in-place writes for updates and insertion. Modern database systems have therefore implemented the Log-Structured Merge-tree (LSM) [1] as their storage layer [2–4]. LSM-trees buffers writes in main-memory and writes them to disk in batches at a later stage, effectively transforming random writes to sequential writes.

A problem with multi-dimensional data is the absence of a natural linear ordering and techniques to improve performance include efforts to reduce the dimensionality of the data, because the memory models of modern computers are one-dimensional [5]. Space-filling curves such as the Hilbert Curve [8] and Z-Order curve [9] have been implemented in R-tree like indices [10, 11] to increase insertion performance and storage utilization.

1.1 Research Questions

The goal of this thesis is to create an experimental LSM-tree data structure which incorporates a R-tree index using space-filling curves and utilizes the LSM-tree properties. By transforming the multi-dimensional geometric objects in a R-tree to one-dimensional objects, it should be possible to incorporate the R-tree to a LSM-tree. Space-filling curves are widely used for reducing dimensionality and different curves are well research for their locality conservation property. There exists database systems that uses R-tree in their LSM-tree structure [2], but the implementation details are hard to come by.

- **RQ1:** Is it possible to utilize the properties of a LSM-tree for a R-tree index?
- **RQ2:** How does it perform?

1.2 Thesis Structure

The thesis is structured into three main parts: The background chapter introduces R-trees, LSM-trees and space-filling curves and their principles, underlying structures and applications. The implementation chapter presents the suggested proof-of-concept implementation, and is followed by results. The thesis then concludes the results and presents potentially further work.

Chapter 2

Background

This chapter focuses on providing some basic background knowledge about the R-tree, space-filling curves and the LSM-tree. Most of the R-tree theory is from [7], while the theory about space-filling curves and LSM-trees are various papers in the literature.

2.1 R-Tree

In recent years, modern services and applications handle more and more multi-dimensional data than ever before. The need for efficient search structures has made the industry recognize the usefulness and necessity of R-trees and its cousins. The original R-tree was described by Guttman [6] in 1984 and set the path for a wide range of alternative access methods being made.

B-trees have been the de facto standard access method in all relational systems for years. While being great for indexing one-dimensional data and doing point-queries, it can not efficiently index spatial data and do range-queries. However, the R-trees are based on B$^+$-tree with support for dynamic organization of d-dimensional data.

The R-tree structure and B$^+$-tree share a similar structure, both are balanced search trees, organizes the data in pages and designed for persistent storage. Instead of using alphanumerical keys, the R-tree uses the minimum bonding d-dimensional rectangles (MBRs) to order the nodes and leaves. Unlike the B$^+$-tree, which has the keys interleaved with pointers, the R-tree has one pointer dedicated to each MBR in the node.

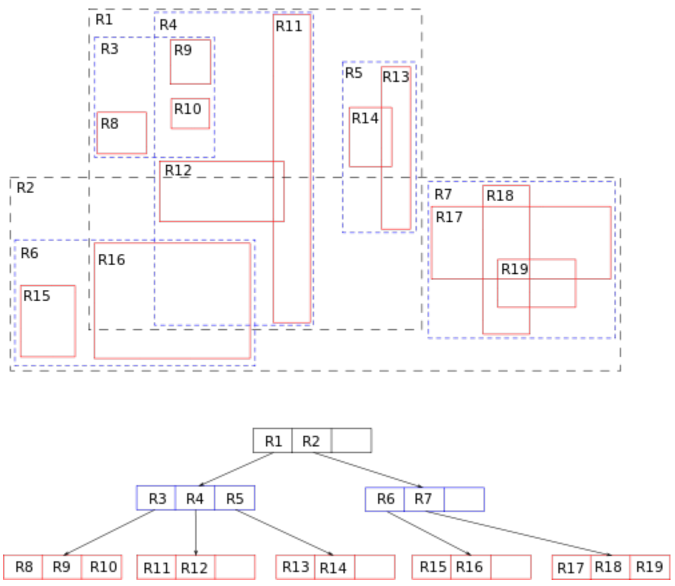


Figure 2.1: Example of a two-dimensional R-tree

A simple example of a R-tree is illustrated in figure 2.1. Every node in the R-tree corresponds to the MBR for its children. For example, $R5$ is the MBR for its leaf node which contains the two entries R13 and R14. Going further up the hierarchy, it is shown that R1 is the MBR for R5 and its siblings.

It is important to note that the MBRs for the nodes at a certain level may overlap and create intersections. Therefore when doing a spatial search on the tree, it may be necessary to visit more than one node before concluding the result of the query. Intersection increases the search space and may therefore greatly influence the performance of search queries. It is also important to examine the candidate objects to resolve false positives. Two objects may have MBRs that intersect, but their geometric shapes do not.

2.1.1 Insertions

The insertion algorithm for the R-tree are very similar to the one for a B$^+$-tree. The traversal starts from the root node and at each level it selects the node with a MBR which require the least enlargement to cover the new entry. If there is a tie, the node whose MBR has the minimum area is chosen. When a leaf node is encountered and it is not full, the new entry is inserted and if enlargement is needed, update the nodes along the path from the root.

In case the encountered leaf node already contains M entries, splitting is required. Since R-trees have d-dimensions, splitting requires a lot more consideration than the splitting in B$^+$-trees. The goal of the split algorithm is to minimize the probability that both the two newly created nodes get invoked during a search query (no overlap). The simplest split algorithm proposed by Guttman is *linear* split, which tries to minimize the total area of both nodes in linear time complexity. Guttman also described two other alternatives, *quadratic* and *exponential*.

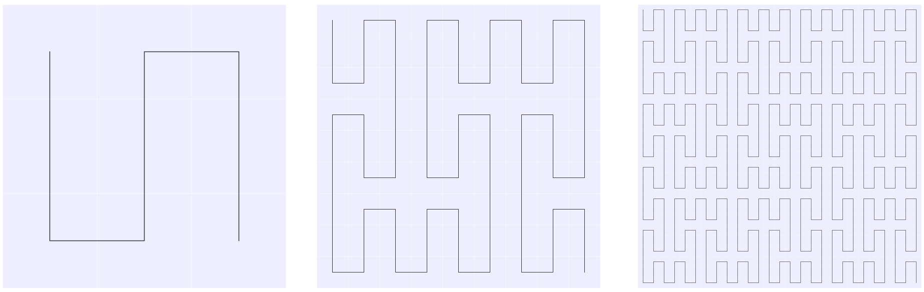
Between different R-trees variants, the insertion and split algorithms are normally the main distinction. A lot of research has suggested better algorithms which either provides better search performance or higher insert throughput. The R*-tree [12] is an example of one variant which is widely accepted as prevailing performance-wise and during insertion considers not only minimization of the area by a MBR, but also overlap, perimeters and storage utilization. Therefore achieving much better search performance.

2.1.2 Search

The main advantage R-trees have over B$^+$-trees are the opportunity to do range queries, which are the task to find all data rectangles that are intersected by a given rectangle Q. The search process starts at the root node and travels to every node whose MBR intersect with Q. When a leaf node is reached, every entry intersecting with Q is added to a temporary set. Finally, all the entries added to the temporary set is examined to determine if the actual geometric object intersect with Q and added to the result set.

2.1.3 Deletions

First step in deleting an entry is to first find the leaf node that contains the element. After the leaf node is found during the search, the element is removed and parent nodes are updated. If the leaf node underflows (fewer than m entries), reinsertion of the remaining elements in the node is usually the recommended method. Reinsertion of elements is considered more appealing than merging nodes because multi-dimensional data does not



(a) First order **(b)** Second order **(c)** Third order

Figure 2.2: First three iterations of the Peano curve

have a natural linear ordering in contrast to one-dimensional data. Additionally, reinsertion maintains the tree quality better after several deletions.

2.2 Space-Filling Curves

Space-Filling curves has its origins as mathematical curiosities, at the end of the nineteenth century. The idea that it exists a continuous one-dimensional curve which passes through every point of a volume or an area, contradicted the established notions in topology at the time [13]. Since it demanded new concepts in topology, many influential mathematicians, such as Hilbert and Peano, studied this new concept. Peano was the first to solve that every point of the unit square could be mapped to a continuous curve [14], but Hilbert was the first to discover the construction of an class of curves with a general geometric procedure [8].

A space-filling curve is a way to create a linear ordering of multi-dimensional data. In simple mathematically terms this corresponds to mapping multi-dimensional data with d-dimensions from indices $\{1, ..., n\}^d$ to one-dimensional sequential indices $\{1, ..., n^d\}$ and vice versa. The first three iterations of the historically first space-filling curve, the Peano curve, is illustrated in Figure 2.2.

One important property for space-filling curves are that the mapping should retain neighbour relations in all dimensions, which also will conserve the locality properties of the data [5]. In reality, creating a sequential order on the data which conserves locality for all dimensions usually fails. This happens because when space-filling curve sequentialize one dimension, it usually fails for the other dimensions [15]. Therefore the preserving of neighbour relations in all dimensions becomes even harder with increasing dimensionality as a consequence of the curse of dimensionality [16].

In the last couple of years the problem of managing multi-dimensional data has gained attention with the development of advanced database systems which require high real-time throughput and efficiency for processing tasks or transactions. Techniques to improve performance include efforts to reduce the dimensionality of the data, because the memory

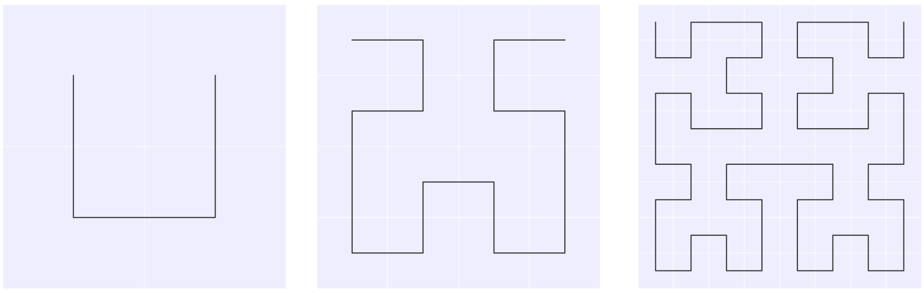

(a) First order (b) Second order (c) Third order

Figure 2.3: First three iterations of the Hilbert curve

models of modern computers are one-dimensional [5]. Examples of application where this mapping is used are; routing, partitioning for supercomputers [17], image processing [18][19], memory management [20], disk scheduling [21], geographical information systems, computing [22] and simulations [23].

2.2.1 Hilbert Curve

The Hilbert Curve is a space-filling curve described by David Hilbert in 1891 [8] and is a variant of the Peano curve. It is widely used in computer science because of its mapping from two-dimensional space to one-dimensional space preserve locality very well [24]. Experimentally it is also shown that the Hilbert curve achieves the best clustering compared to Z-Order and Gray-code curves [25]. Because of the good locality properties Hilbert order is common in proposals for multi-dimensional databases and has been used to increase storage utilization and performance of R-tree indices [11].

The construction and definition of the space-filling curve is based on a recursive procedure which uses reflection and rotation operations to create the next iteration. Figure 2.3 illustrates the first three iterations and the recursive steps. From one iteration to the next, the pattern of the current iterations is copied four times and assigned to four smaller subsquares. The four new copies are connected at their start and end points by using either rotation and/or reflection. It is normally assumed that the curve starts in the lower left corner and ends in the lower right.

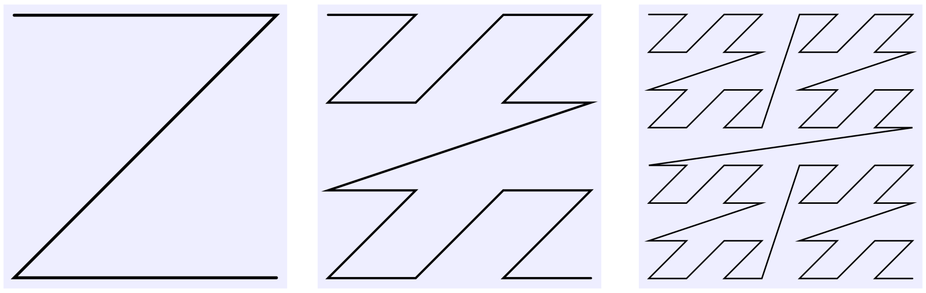
The original Hilbert curve only mapped from two-dimensional space to one-dimensional space but code generation for three-dimensional space has been suggested [26] [27] and a generalized approach for higher-dimensional space [28].

2.2.2 Z-Order Curve

Morton [9] was the first who used the concept of Z-order mapping for linear indexing of 2-dimensional spatial data and the Z-order mapping is therefore usually attributed to

```
px = 0.1010
Py = 0.0111
    ⇓
px = 0.10001000
py = 0.00010101
    ⇓
code = 10011101
```

Figure 2.4: Interleaving bits for a point in two-dimensions



(a) First order **(b)** Second order **(c)** Third order

Figure 2.5: Three iterations of the Z-order curve

Morton. In literature, the Z-order is also referred to as the Peano curve, Morton key order (Morton code) and bit-interleaving. Some confusion may occur, but the Z-Order curve is essentially the Peano curve [25], but are using bit-shuffeling to map multi-dimensional space to one-dimensional space.

The calculation of the z-value of a point in multi-dimensional space is easily done by interleaving the binary representation of its coordinate values. The first step in generating a code for a point in d-dimensions is to take each coordinate and expand it by inserting d-zeroes after each bit. Finally, all coordinates is interleaved to form a single binary number as shown in figure 2.4. Figure 2.5 illustrates three different iterations of the z-order curve in two-dimensional space.

Compared to the Hilbert curve, the Z-order curve is more widely implemented in software. According to [5] one important property for an efficient space-filling curve is that the mapping between data and indices should be easy to compute, which the Z-order curve is. In contrast, the Hilbert curve is relative expensive to compute, but conserves locality far better than the Z-order curve. Faloutsos and Rong's study [29] finds that the Z-order curve had about nearly twice as many sections intersecting with queries as the Hilbert curve sections.

2.3 Log-Structured Merge-Tree

The Log-Structured Merge-Tree (LSM-tree) [1] is an ordered, persistent index structure that supports normal operations such as insert, update, delete and search [2]. LSM-trees provides efficient search and updates by optimizing for frequent and substantial amount of updates. It achieves this by buffering writes and writes the data in batches at a later stage, instead of performing very expensive in-place updates. Batching writes does decrease the efficiency of reads, but the huge increase in data ingestion rates is well worth the penalty.

Today, LSM-trees are used in a lot of different key-value stores where fast indexing of frequent and high-volume updates are required. AsterixDB [2], LevelDB [3], RocksDB [4] and Cassandra [30] are examples of some commercial products that uses LSM-trees or similar variants as its storage layer for structuring storage.

2.3.1 Data Layout

The data layout consists of a hierarchy of storage components (also called levels) with increasing size. The minimum number of components are two, but three or more are common in implementations. Figure 2.6 illustrates a two component LSM-tree. The first component is called C_0 which is located entirely in main memory and C_1 and higher are resident on disk. In some implementations C_0 can also be called for L_0 or *buffer* or some variant of the latter. C_0 is used to buffer updates for higher efficiency since it is stored in main memory and writes are therefore considered free in terms of I/O. However, memory is expensive and limits the size of the in-memory component C_0. Normally, the data structure of C_0 differs from higher components stored on disk which are comparable to a B-tree and data structures such as linkedlist, heap, skiplist, arrays, etc. may be used instead.

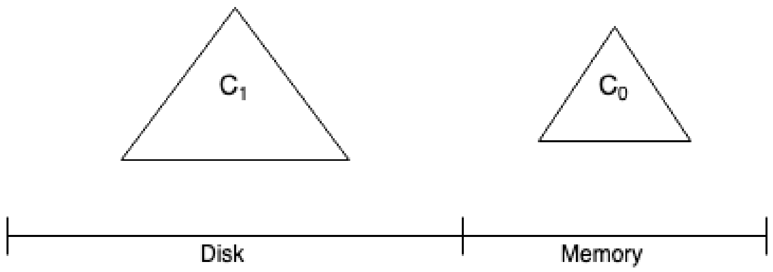


Figure 2.6: Simple two component LSM-tree

If the capacity of C_0 has reached its limits, the content must be sorted on key-order and written to a file on disk. A component's capacity is defined by two properties which defines how the LSM-tree should grow. The first one is a threshold size, which can either be expressed in bytes, number of entries or a factor of the buffer size. The second one, the merge policy, decides when a merge should happen and is a threshold for the maximum number of runs that are allowed for a component.

A run is a collection of one or more files which are sorted on key order and have no overlapping keys. Files are immutable and their format is implementation specific such as SSTable for RocksDB and Sorted tables for AsterixDB. Additional structures like Bloom filters can also be stored in the files to help with search queries by potentially reducing the need for unnecessary I/O.

2.3.2 Writes

Utilizing the main memory makes the writes in LSM-trees very fast. When C_0 is full, the content are written in bulk to the disk and as a consequence transforms huge amount of random writes to sequential I/O. This is very beneficial for both hard disk drives and solid-state drives since they are considerably better at sequential I/O than random I/O. Insert and updates are handle as the same operation in a LSM-tree because both binds a value to a key. Deletion on the other hand is handled the same way except it is marked as an ”anti-matter” record. An ”anti-matter” record will wipe out an older record if encountered. At some point, a component reaches its maximum capacity and the merging process is triggered.

2.3.3 Merging

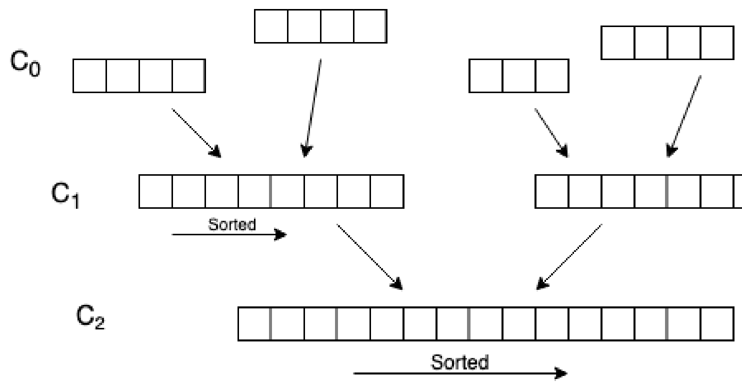


Figure 2.7: Illustrates merging of components

Merging or compaction, is the process of taking a set of sorted files, merging them on key-order and then creates a new run containing the new set of files. A merge process can happen in two different states for a certain component. The first state is when the current component has reached its run limit. Its current content then need to be moved to a higher component to clear more space for new runs. It is normal to merge all runs in a given component, but it is possible to use different strategies to decide which and how many runs to move upwards and merge. The second state that triggers a merge process, happens when a run is moved upwards, but the destination component has no more room for more runs. Therefore the run must be merged with an already existing run. As a consequence, merging can cause a cascading effect of further merges through higher components.

Figure 2.7 illustrates the merging in a LSM-tree. The simplest implementation is to merge one run at a time upwards to the next component. When C_0 has reached its threshold, all the content of one run is selected and flushed to the next component C_1. During the merge, only the newest value for a key is kept if update and delete operations were performed. A cascading effect can occur and a run in C_1 is pushed upwards to C_2.

2.3.4 Lookups

Lookups and searches is the most expensive operations in a LSM-tree since with increasing number of disk components, more and more disk access is required and the performance degrades [2]. Therefore extra structures are often used to optimize search queries. Bloom filters are used to increase the speed of point-queries in a run and can either be persisted to disk for a run or rebuilt during startup. Fence pointers are also applied in some cases, which allows for skipping runs not in a relevant range.

Point-queries are performed in a propagating manner. The search start in C_0 and if no matching key is found, the process continuous to the next component C_1 and so on until a match is found. Range-queries on the other hand must search every component and must be wary for update and delete records during the search.

2.3.5 Component Management

The concept of a component are defined and managed by the LSM-tree implementation, while files are controlled by the file system. Maintaining a consistent view of immutable files is crucial for guaranteeing readers query correctness. It is therefore important to have some sort snapshot management with catalog or list structures for referencing files for all components persisted on disk and those in main memory. Managing snapshots ables the merging tasks to work in the background without disturbing other operations.

Chapter 3

Related Work

This chapter presents related work in the literature that are very relevant for this thesis and its proof-of-concept data structure. This includes the Hilbert R-tree which uses the Hilbert Curve and AsterixDB which uses a LSM-tree base storage layer with different indices incorporated into the LSM-tree.

3.1 Hilbert R-tree

The Hilbert R-tree [11] is a R-tree variant which is very similar to the B$^+$-tree. Actually, it is a B$^+$-tree with R-tree characteristics, but includes support for multi-dimensional geometrical objects. It achieves this by using the Hilbert space-filling curve to calculate a Hilbert value for the entries' geometrical object, in addition to calculating the MBR. Nodes in the tree are then given the largest Hilbert value (LHV) as an additional property. Figure 3.1 illustrates an organized Hilbert R-tree.

Range queries are processed the same way in the Hilbert tree as the original R-tree, checks if MBRs intersect, but point queries may utilize the Hilbert value instead of the MBR. However, the significant difference lies in the insertion procedure compared to other R-tree variants. Instead of using the MBR as a determining factor when comparing, the Hilbert value is utilized.

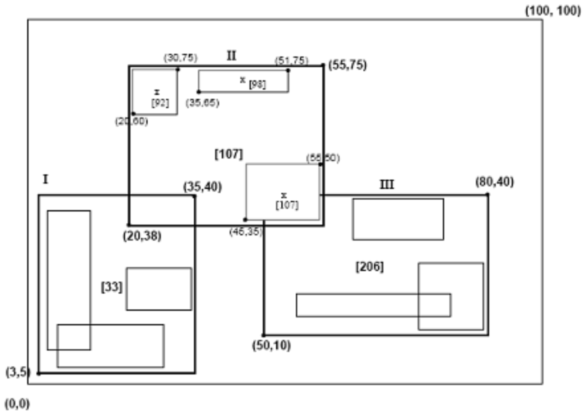


Figure 3.1: Hilbert R-tree organization (Hilbert values in brackets), from [31]

The traversal during insertion starts in the root node and at each level the Hilbert value H for the new entry is compared to the largest Hilbert values to the nodes. It then selects the node with the smallest Hilbert value that is larger than H. Or if no such nodes exists, the node with the largest Hilbert value is selected. If a leaf node is encountered, the new entry is inserted and all nodes along the path are updated to reflect changes in MBR and/or largest Hilbert value.

Since the Hilbert space-filling curves creates a sequential ordering, nodes and entries have a well-defined order. This means that the order of entries at the leaf level always will be the same regardless of the order new entries are inserted. As a result, it is possible to delay node splitting when a node overflows and instead move entries to sibling nodes. If the sibling nodes also have reached their capacity, one new node is created and all entries are distributed evenly among them. Delaying the splitting of a node increases the storage utilization considerably compared to the original R-tree.

However, the Hilbert R-tree does not perform well with high-dimensional space because of the properties of space-filling curves, as discussed in section 2.2.

3.1.1 Packed variant

The Hilbert packed R-tree [32] is based on the structure of the Hilbert R-tree and is designed for static data. The sequential ordering of entries with the Hilbert space-filling curve is exploited to build a Hilbert-tree with almost 100% storage utilization.

Dynamic R-trees are normally constructed from the top and downwards, while the Hilbert packed R-tree is constructed in a bottom-up approach. The packing algorithm goes as follows; First the Hilbert value of all entries are calculated and sorted accordingly. While there are remaining entries, the next $f(\mathit{fanout})$ entries are assigned to a new node. This is repeated for every level until a level only contains one node, which is assigned as the root of the tree. After the procedure has completed, all the nodes in the tree are full, except from the last node at every level.

3.2 AsterixDB's Storage Management Layer

Apache AsterixDB [33] is a parallel, semi-structured information management platform that provides the ability to ingest, store, index, query, and analyze mass quantities of data [34]. It uses ideas from semi-structured data, parallel databases and data-intensive computing to create a platform that runs on shared-nothing clusters.

Inside AsterixDB lies maybe the most important part, the LSM storage layer. The storage layer contains a generalization for converting a class of indices such as B-trees and R-trees, so they can be used in a LSM-tree. The framework provides basic operations such as insert, delete, and bulk-loading for the converted indices and reduces the time and effort for constructing a index structure from scratch.

The LSM storage layer consists of four different indices, a primary LSM B-tree, a secondary LSM B-tree, a secondary LSM R-tree and a secondary LSM keyword Inverted Index. A overview of the storage layer and example queries are shown in figure 3.2.

The primary LSM B-tree has different structures for the in-memory component and disk persistent component. The in-memory component consists of a single B-tree with entries are ordered by their primary key. The disk component is similar except it also contains a Bloom filter to reduce the search space during point-queries.

The secondary LSM B-tree does not have a Bloom filter on the disk component and the usually (key, value) pair is instead on the form: (seconday key, primary key). This provides an opportunity to have extra indices on fields containing integer data.

Since a B-tree is only suited for one-dimensional data and point-queries, AsterixDB uses a LSM R-tree for fields with multi-dimensional data. While the in-memory component is a normal R-tree, the disk component uses a Hilbert curve to order the entries. The Hilbert

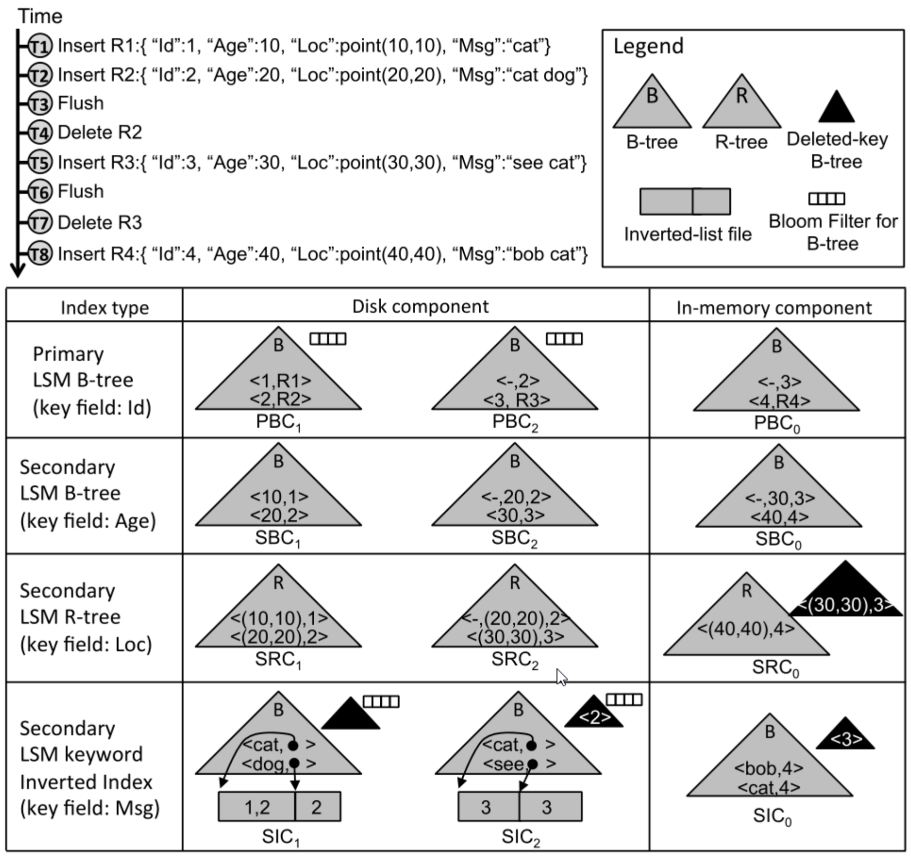

Time
T1 Insert R1:{ "Id":1, "Age":10, "Loc":point(10,10), "Msg":"cat"}
T2 Insert R2:{ "Id":2, "Age":20, "Loc":point(20,20), "Msg":"cat dog"}
T3 Flush
T4 Delete R2
T5 Insert R3:{ "Id":3, "Age":30, "Loc":point(30,30), "Msg":"see cat"}
T6 Flush
T7 Delete R3
T8 Insert R4:{ "Id":4, "Age":40, "Loc":point(40,40), "Msg":"bob cat"}
Legend
B-tree
R-tree
Deleted-key B-tree
Inverted-list file
Bloom Filter for B-tree
Index type
Disk component
In-memory component
Primary LSM B-tree (key field: Id)
<1,R1> <2,R2> PBC1
<-,2> <3, R3> PBC2
<-,3> <4,R4> PBC0
Secondary LSM B-tree (key field: Age)
<10,1> <20,2> SBC1
<-,20,2> <30,3> SBC2
<-,30,3> <40,4> SBC0
Secondary LSM R-tree (key field: Loc)
<(10,10),1> <(20,20),2> SRC1
<-,(20,20),2> <(30,30),3> SRC2
<(40,40),4> <(30,30),3> SRC0
Secondary LSM keyword Inverted Index (key field: Msg)
<cat, > <dog, > 1,2 2 SIC1
<2> <cat, > <see, > 3 3 SIC2
<3> <bob,4> <cat,4> SIC0

Figure 3.2: AsterixDB's storage layer and operation example, from [34]

curve creates a linear ordering of the entries such that it is possible to do incorporate the R-tree index into the LSM-tree. Insert operations is handle the same way as a traditional R-tree, but deletes are added to an in-memory B-tree. When the in-memory component is flushed to disk, entries in the R-tree and B-tree are merged, but the B-tree entries changes to anti-matter.

The last secondary index is the Inverted index which is used to index keywords on string fields. The in-memory component consists of a B-tree and an extra B-tree for deleted keys. The on-disk component share the same structure, but also consists of a Bloom filter for optimizing search queries. In contrast to the LSM R-tree where deleted keys are merged into the main structure during flush operation, the Inverted index merges the deleted keys separately and retains its own tree structure on disk, as illustrated in figure 3.2.

Chapter 4

Implementation

This chapter describes the ideas and implementation of an experimental LSM-tree data structure that incorporates a R-tree index in the disk component. First an overview of the implementation is presented and is then followed by the actual implementation explained. The basic structure is explained first followed by the internal components with their corresponding challenges and ideas, before moving up to higher components again. In the end, a proof-of-concept structure is designed and constructed.

4.1 Overview

This thesis' purpose is to propose a proof-of-concept structure for a high throughput LSM-tree that incorporates a R-tree index in the disk component. This includes challenges such as how to merge-sort components into a higher component while maintaining an efficient R-tree structure, which compaction algorithm to apply and how to sequentialize multi-dimensional data.

Two methods for performing the merge-sort process was implemented, one is based on the packing algorithm for Hilbert Packed R-tree and the second one is an iteration on the same concept, but optimized for caching. Compaction uses a leveled-N algorithm inspired by Fluid LSM [35] with less write amplification than leveled. The last main part of the implementation is the usage of space-filling curves such as Hilbert Curve and Z-Order curve for sequentializing multi-dimensional data.

4.2 Structure and Files

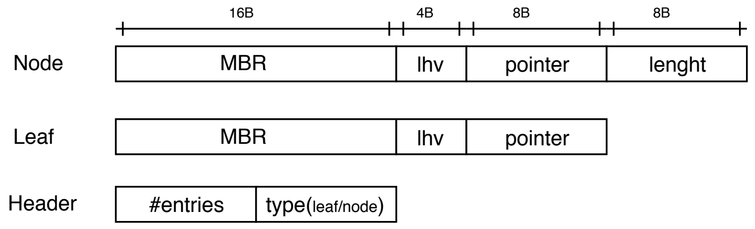


Figure 4.1: Data layouts for internal objects

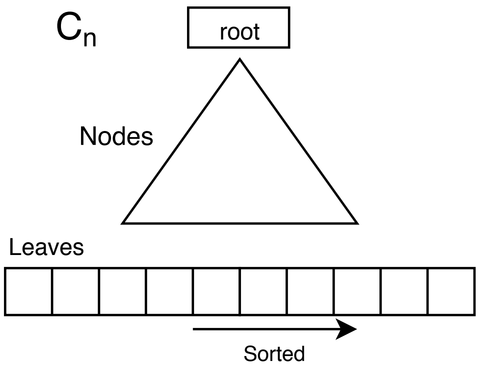


Figure 4.2: Disk component

The first thing to consider when incorporating a R-tree into the disk component for a LSM-tree, is a proper structure. The first property to note is that the entries must be sorted and stored in a sequential way, as required by the properties of the LSM-tree.

Figure 4.1 shows the data layout of the three different types of objects stored in the files. The smallest one, the header, is just a marker for a sequence of entries. It specifies the total length in byte for the sequence of entries and specifies if it is either a sequence of leaves or normal nodes. The other two objects, leaf and node, contains a MBR, the largest hilbert value (lhv) or an equivialent and a pointer to its children location. They are are based on the same layout suggested in the paper [6], except that the node object also contains the total length in bytes for its children. Together, the pointer and length attributes determines where the children are located.

A potential structure for a run in a disk component C_n for a LSM-tree is illustrated in figure 4.2. It is composed of three different parts with each part stored as separate files. The first part is the leaf level and contains all entries and resembles the normal LSM structure. Entries are sorted in ascending order by their corresponding space-filling value, which may either be a Hilbert value or Morton code (Z-order) in this implementation.

The second part is the actual tree structure. This is the upper levels for a R-tree resembling the Hilbert R-tree and contains only nodes. Section 3.1 covers this in more detail. The third and last part is the “root” part, it contains metadata about the run such as size, number of entries, leaves location or other data that may be useful in some situations.

The reasoning for storing the parts in different files is that it may be useful for applying more aggressive caching techniques. The “root” part should mostly be available in main memory since it contains useful metadata for merging or other purposes. During a merging process, only the leaves are of interest since it is not possible to merge the upper levels on key-order since the MBRs need to be recalculated. More details are about the merging process are covered below in section 4.4.

4.3 Space-Filling curves

The LSM-tree requires that entries can be sorted in key-order for the purpose of being able to do the merging efficiently. This poses a challenge for multi-dimensional data that does not have a linear ordering defined in its properties. To overcome this, space-filling curves are utilized to sequentialize multi-dimensional geometric objects. More specifically, the Z-order curve and the Hilbert curve have been implemented and is used during insertion, where the entry’s space-filling curve value is calculated from its MBR. As discussed in section 2.2, they retain their neighbour relations and conserves locality, which is important for indices.

Implementing the calculation of a Hilbert value is non-trival and therefore an external library was used to calculate the value. Z-order curve is a lot simpler and can be done by interleaving bits, either by table lookup, multiplication or magic numbers. Figure 4.3 shows how the calculation of the Morton code for a two-dimensional point can be done using bit interleaving and binary magic numbers. The implementation in this thesis is based on the code in figure 4.3, but extended it to 64-bits from 32-bits. By using 64-bits it is possible to calculate the value for a point represented by two floats (32-bits) without losing much precision and reducing the chance of collisions with keys.

```
unsigned int B[] = {0x55555555, 0x33333333,
                    0x0F0F0F0F, 0x00FF00FF};
unsigned int S[] = {1, 2, 4, 8};

unsigned int x;
unsigned int y;
unsigned int z;

x = (x | (x << S[3])) & B[3];
x = (x | (x << S[2])) & B[2];
x = (x | (x << S[1])) & B[1];
x = (x | (x << S[0])) & B[0];

y = (y | (y << S[3])) & B[3];
y = (y | (y << S[2])) & B[2];
y = (y | (y << S[1])) & B[1];
y = (y | (y << S[0])) & B[0];

z = x | (y << 1);
```

Figure 4.3: Interleave bits by Binary Magic Numbers

4.4 Merging

Merging in a LSM-tree is perhaps the most difficult task to tune for high write throughput without interfering with other queries. This thesis does not cover the concepts around reading from a LSM-tree, but explores the challenges with write throughput. Therefore are constrains for accessing runs for read operations only partly considered in the choice of compaction algorithm.

There exists a lot of different compaction algorithms for merging components in a LSM-tree. Some can be very complex to implement, tiered compaction, and others are more straight forward, leveled. Since compaction is not in the focus of the thesis, a leveled-N algorithm is implemented. It differs from leveled with that it allows for more than one run per component. It also provides less write amplification and more read amplification than leveled which was introduced in the LSM-tree paper by O'Neil et al [1].

When the merging progress is triggered for a given component C_n all its runs are merged into one new run in component $C_n + 1$, which are similar to leveled compaction. This means that the component $C_n + 1$ is many times larger than C_n, since all runs in one component are merged. While all non-max components have n runs, the max component has only one run. This is similar to the compaction algorithm Fluid LSM, suggested by Niv Dayan and Stratos Idreos [35]

While a compaction algorithm defines how runs and components should interact, the actual merging of two or more runs is implementation specific. The structure suggested in section 4.2 poses some challenges and two different strategies are suggested and implemented.

4.4.1 Partly

The first strategy are one called “partly” in this thesis, it is similar to the packing procedure used for the Hilbert Packed R-tree [32]. The procedure is illustrated in figure 4.4 and goes as follows:

The first step is to gather all the *leaf*-files from the lower component’s runs. The previous tree structure in the lower component is ignored simply because it is not possible to merge without significantly degrading the R-tree’s performance and storage utilization. Then the files are merged using a normal merge procedure and written to disk sorted by their space-filling curve value in ascending order.

After the leaves are sorted and written to disk, the tree construction begins. A new node is created at level l and assigned leaves in ascending until the fanout is reached, its MBR and lhv is updated accordingly. This is repeated until all leaves are assigned to a node. Then the same steps are repeated in level $l - 1$ for the new nodes until a level only contains a single node. Then this node becomes the root of the tree.

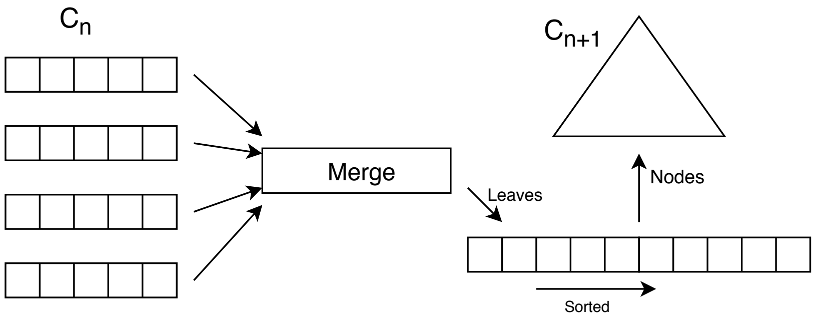


Figure 4.4: Illustrates “partly” merging of components

4.4.2 Continuous

The second strategy is referred to as “continuous” and tries to optimize the tree construction in the partly procedure. The hypothesis is that the requirement to read the entire leaf-file after it is written to disk for the partly procedure, may decrease write throughput for very large runs when the leaf-file does not fit in main memory.

The change is quite simple; Instead of constructing the R-tree after the merge is done, the tree is constructed continuously while merging. This means that it is not necessary re-read the leaf-file. Figure 4.5 shows this process. During the merging a new node n at level l is created and is filled with sorted entries until fanout is reached. Then node n is assigned to a node at level $l - 1$ and a new node is created at l, their lhv and MBR are updated accordingly. This process cascade upwards when a node reaches its fanout and terminates when the runs are sorted. In the end, the entire R-tree index is constructed and no extra read operations are required.

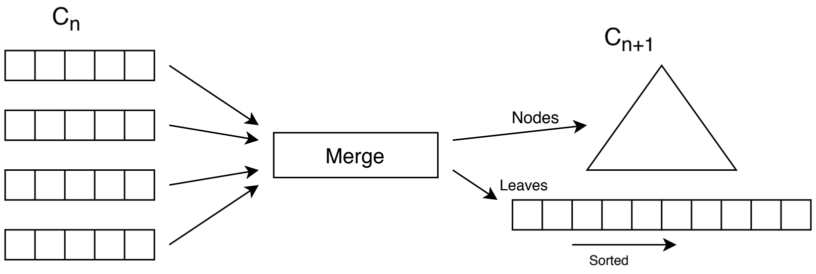


Figure 4.5: Illustrates “continuous” merging of components

4.5 Minor considerations

This section addresses some implementation specific challenges with LSM-trees.

4.5.1 Threading

In this day and age, parallel programs have become more important because the advancements of parallel systems. In this thesis, threading is only implemented on the buffer component and the merging of the buffer component to C_1. The advantages threading gives at the higher components are not too significant because the merging process is for the most part heavily I/O bound. Every run in the buffer is run in parallel and has its own thread. This means that it is possible to handle multiple updates at the same time. When a run in the buffer has reached its threshold, it is sent to a different thread for merging the content to disk.

Further work one should consider researching methods for task-parallelism and data-parallelism for the merging process of higher components. It may increase write-throughput if the task is not to heavily I/O bound. Also for range-queries, data-parallelism could be interesting. Searching multiple runs in parallel in a component could increase performance significantly.

4.5.2 Buffer

The first component in a LSM-tree is C_0 or sometimes called the buffer and is stored in main memory. Its main purpose is to buffer inserts and updates for efficiency and batch to disk later. Often its structure is different from the disk components since it is only intended for in-memory operations. Structures such as heaplist, arrays and skiplists can be used. In this implementation, a normal array is used for the buffer for its simplicity for write operations. An array will probably perform very poorly on range-queries which is normal on R-tree indices. For further work where read operations are considered, a dynamic R-tree variant based on the Hilbert R-tree should most likely be used instead since it is more trivial to do range-queries.

4.5.3 Snapshots

In this thesis read operations are not considered on the implemented LSM-tree and therefore it does not have implemented a procedure for maintaining a consistent view of the components' files. Usually some sort of global catalog structure is stored in-memory for accessing a snapshot of the current LSM-tree without getting disturbed by write operations.

Chapter 5

Results

The following chapter present and compares results from different configuration for the proof-of-concept LSM-tree that is implemented in this thesis. Results are evaluated in terms of completion time, reads over time and total number of reads. Writes are not considered since for an input set with fixed size it will always be the same.

5.1 How testing was performed

The different test were run several times and the median with regards to total runtime was selected for data analyze to remove outliers. The median is chosen over mean because sometimes an outlier performed significantly better or worse than the other runs and therefore would have skewed the result in a calculated average.

The tests were performed on commodity hardware; an i3-8100 with 16GB RAM and two SSDs striped together in ZFS. More accurate details of the setup are described in the sections below.

5.1.1 Default configuration

The default parameters for a test run are shown in table 5.1. If nothing else is specified in a section, the default parameters apply.

Parameter	**Value**	Description
Buffer size C_0	1048576	Threshold for a run in C_0
Buffer threads	2	Number of input threads aka runs in C_0
Merge threads	2	Number of threads merging C_0
Fanout	8	
Curve	Z-Order	Space-filling curve used
Runs	1	Runs at C_n, C_{n-1} have two times more runs etc.
Levels	4	Max number of components
Merge Strategy	Continuous	
Read buffer size	32k	
Write buffer size	32k	

Table 5.1: Default parameters

5.1.2 System

System configuration
Intel(R) Core(TM) i3-8100 CPU @ 3.60GHz
16GiB DDR4 2400MHz
Crucial MX500 500GB x2
Debian GNU/Linux 9 (stretch)
Linux 4.9.0-7-amd64
openjdk 11.0.3 2019-04-16

Storage
The storage consists of two MX500 SSDs, which are striped together in a ZFS pool. This

was done to not be I/O bound in some scenarios.

Notable ZFS properties
recordsize: 128k
blocksize: 4k
ashift: 12

5.1.3 Dataset

There are many ways to generate datasets. One could use synthetic ones or real data that are collected. Different data distributions may perform poorly on some indices and better on others. Therefore are usually tests done with different types of distributions such as uniform distribution and clustered datasets.

In this thesis, the dataset is generated with uniform distribution, since the distribution does not matter for performance in this implementation of a LSM-tree. This is because the R-tree index is always reconstructed when inserting new entries, as explained in section 4.4. The dataset contains a total of 100 million entries with location data as points (longitude and latitude) and a random *long* value. Each entry has a size of 32 bytes. Before the the insertion process starts, the whole dataset is loaded into memory to reduce any influence read operations could have on the LSM-tree.

5.2 The art of runtime evaluation

A paper written by Krieger et al. [36] discusses and evaluates pitfalls and challenges with testing the performance of implemented algorithms. It is an interesting read and they suggest that there should be payed more attention to runtime experiments and should be handled with care. They also have a lot of recommendations for doing experiments. In short, the first recommendation is to find the fastest implementation. The second recommendation is to use proper parameters and datasets, which is not chosen to skew the results. This is to ensure more fair and conclusive comparisons in the literature.

To do experimental evaluations is not trivial at all and the results in this thesis should be used as guidelines for potential further work and not directly compared to other works.

5.3 Fanout

Fanout is the number of children in internal nodes and could have significant impact on the performance of a R-tree. When the fanout is increased, the tree height is reduced and results in fewer disk access for point- and range-queries. It could also affect write performance since a higher tree requires additional levels to be created or updated.

As shown in figure 5.1, after a fanout of 2, the number of operations quickly stabilizes for both reads and writes. There seems to be a slight reduction towards a fanout of 32, but is probably in the margin of error. It is to be expected that fanout does not have a huge impact on writes since the reduction of internal nodes are negligible compared to the size of the leaf level.

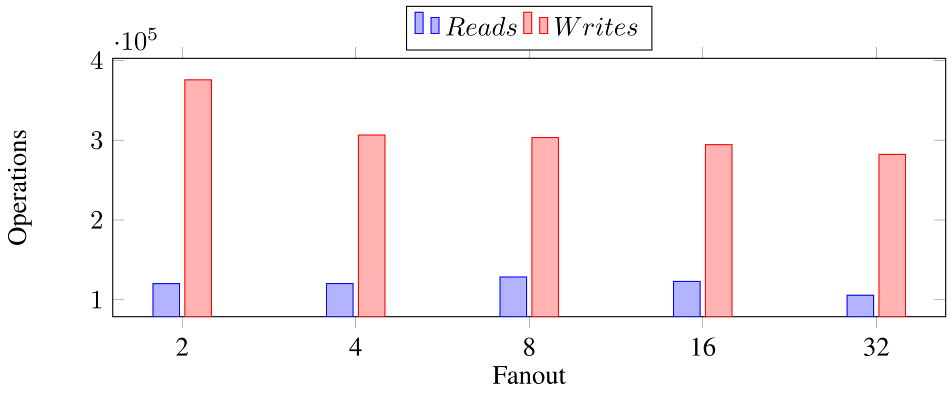


Figure 5.1: Total reads for different buffer sizes

5.4 Buffer size

Testing buffer size is not too relevant for the suggested implementation, but it can be important to tweak the value for optimal write throughput. The total number of reads for different buffer sizes are shown in figure 5.2. This shows that there are huge differences in the number of reads for buffer sizes. About a 40% increase in reads from the lowest ones to the larger ones. The jumps are also very noticeable between buffer sizes.

The interesting part is that $4k$ is the block size of the SSDs used during testing. So the high reads from values below that is expected, but then the number of reads jump by a lot at $8k$ and $16k$ and falls again at $32k$. The fall at $32k$ to $256k$ is probably due to the limits of the L1 and L2 cache. For the processor used in testing, the L1 cache has a size of 32kB per core and L2 has a limit of 256kB.

If instead the actual runtimes shown in figure 5.3 are considered, it shows that there are not a clear correlation between runtime and number of reads for different buffer sizes. After $4k$ there are no significant performance loss for different buffer sizes. For some workloads

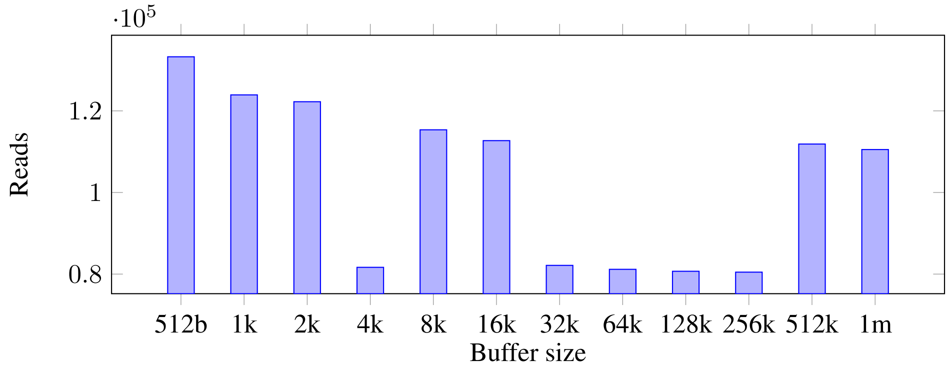


Figure 5.2: Total reads for different buffer sizes

this behavior may be different and testing and tuning should probably be done for different types of systems.

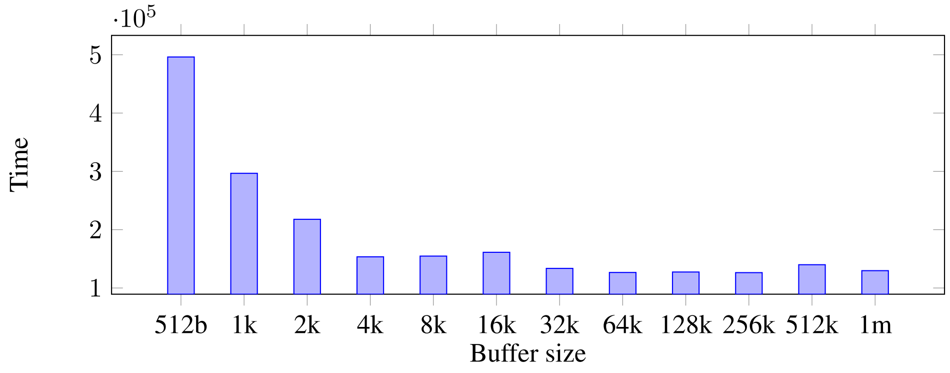
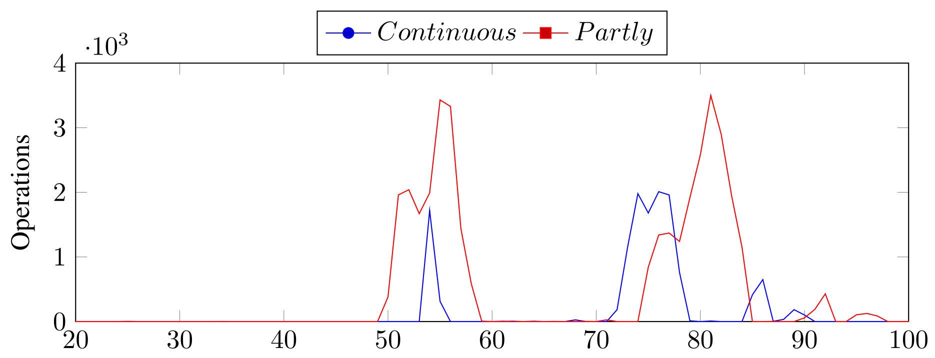
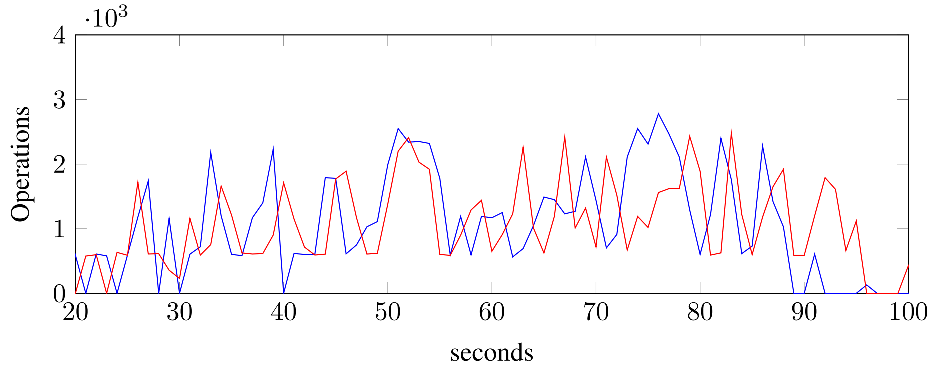


Figure 5.3: Runtime for different buffer sizes



(a) Reads



(b) Writes

Figure 5.4: Operations over time for partly and continuous merging

5.5 Merging

In this thesis, two different merging strategies were suggested in section 4.4, "continuous" and "partly" merging. Two different scenarios of tests were performed to compare the number of reads for both. The first scenario is in a environment were the merging is not limited by memory. This means that most of data needed during the merge process can be stored in-memory and is cached by the operating system. In the other scenario, it is heavily memory limited and only parts of needed data are cached in main memory by the operating systems. Limiting memory is done by artificially amplifying the entries size by a factor of 4 and quadruples the memory requirement. This may be the most realistic scenario for a high throughput service.

The results from the first scenario are shown in figure 5.4a for reads and figure 5.4b for writes. There are no significant deviations for writes between the two merging strategies, but "continuous" finishes the run slightly faster at the ~90 seconds mark, while "partly" finishes at about the ~100 seconds mark. Continuing to number of reads shown in 5.4a, there are some higher and wider spikes for "partly" and probably contributes to the increased runtime.

In the second scenario, shown in figure 5.5a and 5.5b, the effects from the first scenario is amplified. The spikes for reads in figure 5.5a for "partly" is more extreme than previously and the effect on writes is also visible in figure 5.5b.

Figure 5.6 shows the total runtime for both strategies in both scenarios. It is clear that the "continuous" strategy is significantly faster when memory is limited. In figure 5.7 the total number of reads and writes for both scenarios are presented and "continuous" has less than half the reads than "partly".

In total, these results live up to the hypothesis covered in subsection 4.4.2. It is shown that the "continuous" merging has a significant impact on the number of reads during the merge process and can reduce the runtime by a little bit.

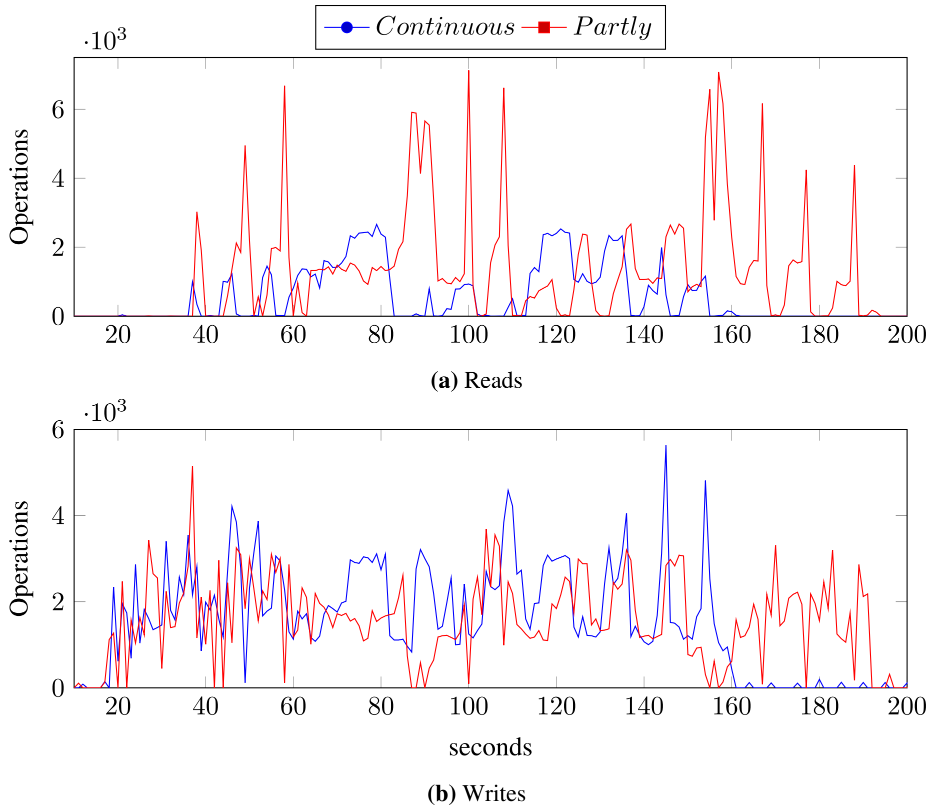


Figure 5.5: Operations over time for partly and continuous merging with limited memory

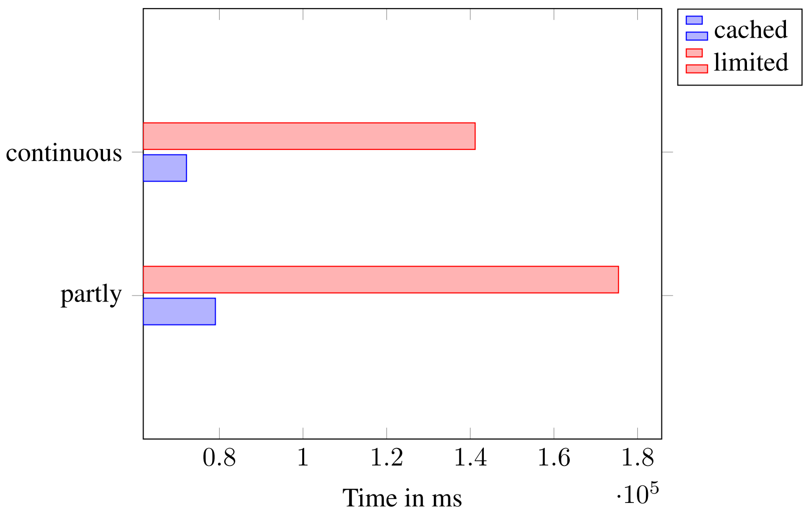


Figure 5.6: Comparison for total time spent for merging strategies

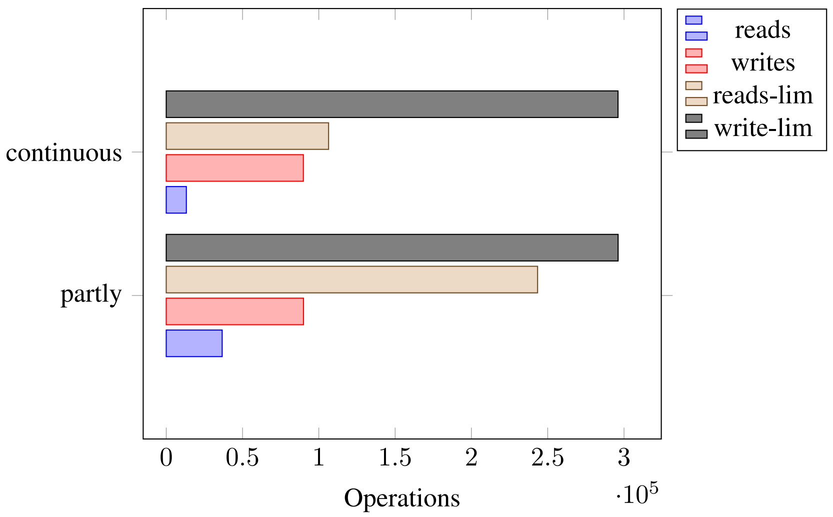
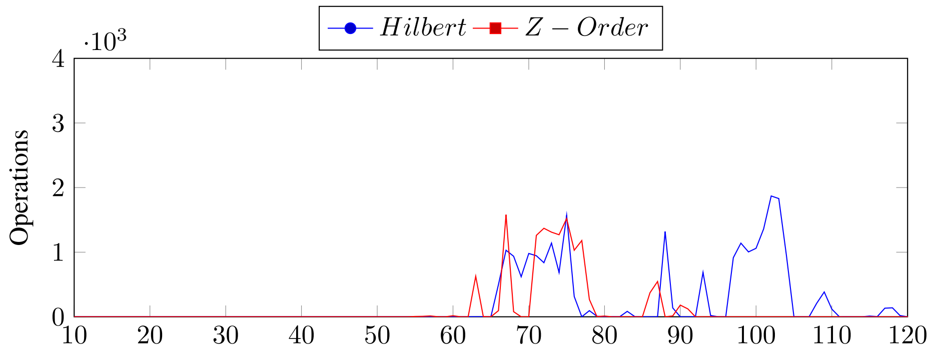


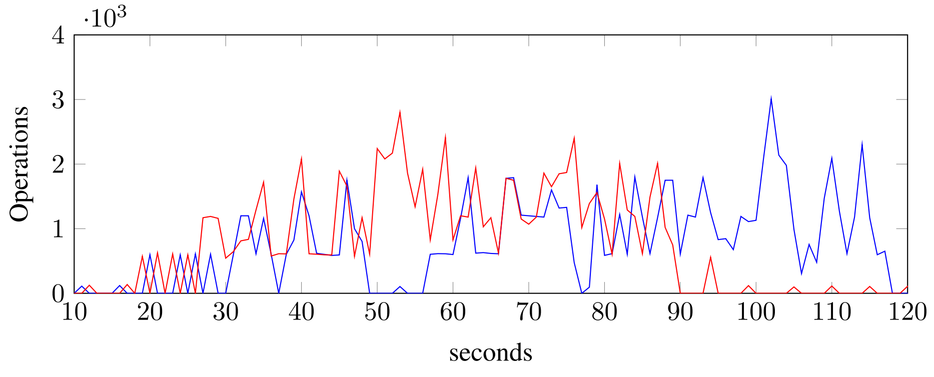
Figure 5.7: Comparison for total operations for merging strategies

5.6 Z-order and Hilbert-curves

The testing of Z-Order curve and the Hilbert curves was done in two different scenarios. The first run was with no amplification of entry size to not to be limited by memory. The second run had an amplification of a factor of four. This means that the entry size were increased during writing to simulate a high throughput scenario without increasing the memory requirement to hold the test dataset.



(a) Reads



(b) Writes

Figure 5.8: Operations over time for Z-Order and Hilbert Curve

In figure 5.8, we can see that there are significant less operations for the Z-Order curve compared to the Hilbert Curve. Z-Order Curve also finished earlier, around 90 seconds while the Hilbert Curve used a total of 120 seconds on the same dataset. This scenario was also not memory limited and therefore the application was for the most part not I/O bound. This means that a reduction in total required processor cycles, gives better throughput.

If we look at figure 5.9, which was a memory limited run, was more I/O bound that the scenario above. This is shown by a reduced difference between the two space-filling curves. Both write and read graphs for operations are similar for Z-Order and Hilbert, while Z-Order have slightly higher throughput and finishes in less time.

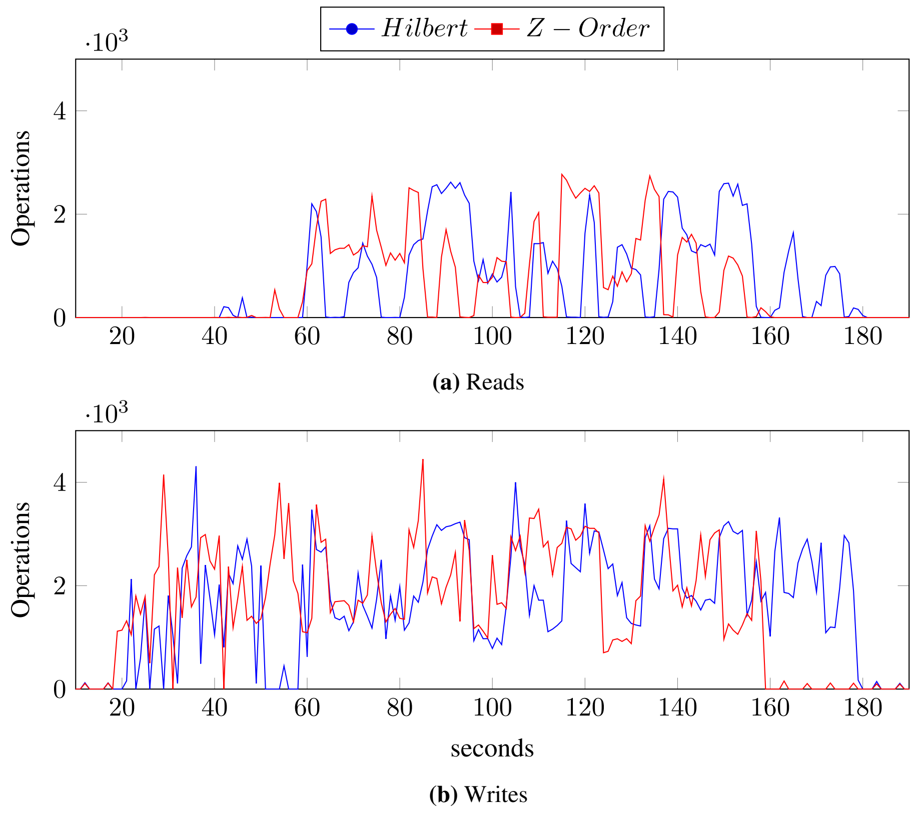


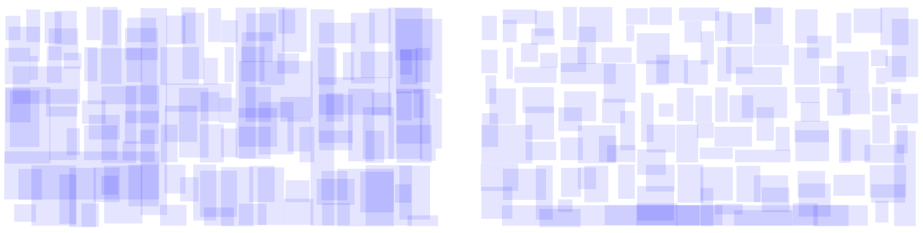
Figure 5.9: Operations over time for Z-Order and Hilbert Curve with limited memory

Pure throughput is not usually the deciding factor in a R-tree. The degree of overlap and intersections in the final tree can have huge impacts on query performance. One of the upper levels of the R-tree index is shown in figure 5.10 from the first scenario. As expected, the overlap between nodes for a Z-Order curve in (a) are of a higher degree than the Hilbert Curve in (b).

5.7 Threading

Threading has not been a major concern in this thesis, but some test were still ran. The results are displayed in figure 5.11 and shows the runtime for different thread configurations. The test were performed on a four-core processor and therefore has some limitations for multi-threading. bt stands for *buffer threads* and are the number of threads handling arriving entries, while bmt is the number of threads that merges a buffer (C_0) to component C_1.

By having more than two buffer threads, it seems like starvation is occurring and is prob-



(a) Z-Order curve **(b)** Hilbert Curve

Figure 5.10: Comparison of overlap between Z-Order and Hilbert

ably due to other threads blocking and the limited number of cores. The number of buffer threads seems to not be the limiting factor, but instead the merging of buffer components to C_1 have a larger impact on throughput.

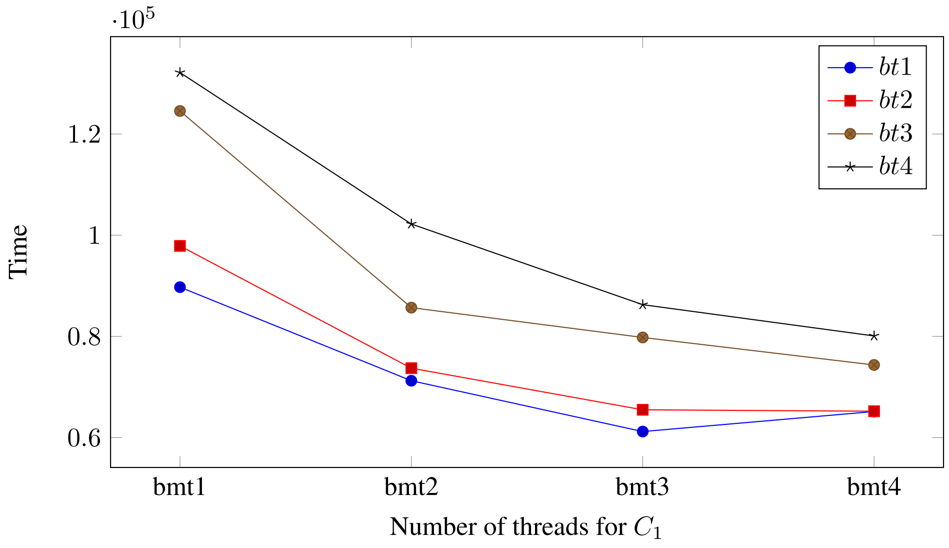


Figure 5.11: Run time for buffer with n-threads with m-threads for C_1

Chapter 6

Conclusion and Further Work

In this chapter the research questions presented in Section 1.1 are evaluated. The implementation is discussed and potential improvements and considerations are suggested for further work.

6.1 Conclusion

The purpose of this thesis was to create an experimental LSM-tree data structure that incorporated a R-tree index with the usage of space-filling curves to cope with the large amounts of data generated in modern applications and services. We presented two research questions in Section 1.1 that specified the goal of the project.

- **RQ1:** Is it possible to utilize the properties of a LSM-tree for a R-tree index?
- **RQ2:** How does it perform?

Throughout the thesis we have answered **RQ1**. In Chapter 4, a proof-of-concept LSM-tree with a R-tree index was created and achieved a positive result for **RQ1**. Space-filling curves was utilized for the mapping of multi-dimensional data to one dimension and two different curves were implemented; the Z-Order Curve and the Hilbert Curve. Z-Order curve was the simplest one to implement which could be done with bit-interleaving. By fulfilling the LSM-tree requirement of sequential data, it was made possible to use the LSM-tree structure for a R-tree since multi-dimensional data could be sorted in a linear order.

The biggest challenge proved to be the merging of runs in the LSM-tree. The nature of the R-tree makes it hard to construct efficiently. Based on an algorithm from Kamel et al. [32], two different merging strategies were suggested in this thesis. Both used the concept of reconstructing all of the upper levels of the R-tree during a merging process to create a new efficient index with high space utilization.

In light of **RQ2** some performance tests were performed. For the two different space-filling curves, Z-Order and Hilbert Curve, we saw that for raw throughput in both a I/O bound scenario and not, the Z-Order curve was better. The differences in throughput was only slightly reduced for the Hilbert Curve compared to the Z-Order curve in I/O bound scenarios. The difference of overlap between internal nodes was only shown briefly in figure 5.10.

The results from the testing of the two merging strategies, “partly” and “continuous”, discovered that “continuous” was superior to the “partly” strategy. By removing the requirement in “partly” to re-read the leaf level after merging to construct the R-tree, the number of read operations was cut in half for “continuous” as shown in figure 5.7.

In addition to the space-filling curves and merging strategies some minor tests for buffer size, fanout and threading was done. Fanout proved to not be very important for write throughput while buffer size and threading showed more potential. Choosing the right buffer size could be critical for fine tuning write and read performance.

To summarize; **RQ1** proved to be possible with some simple design chooses, but requires further testing and considerations. It should also be tried to be implemented in a real database system such as AsterixDB or RocksDB, this will also give **RQ2** more meaningful data.

6.2 Further Work

6.2.1 Merging

More research should probably be done with the merging process, since the merging of R-trees are non-trivial. The two methods suggested in this thesis provides an alright foundation to extend and derive into better methods.

It would also be interesting to look at merging of only runs with non-overlapping key-ranges. This could make it possible to reuse the already constructed tree structure from the runs instead of reconstructing it every time. This will save a lot of CPU time and I/O operations, but is maybe only feasible for lower components since they usually have more runs. Could also have multiple merging procedures with different properties and choose the one which are best for a specific set of runs.

Multi-threading is an area of interest and a parallel the merging technique could further improve throughput, but may not be too trivial for a R-tree. Methods to either do it with data-parallelism or task-parallelism would provide better utilization of today's modern highly parallel hardware.

6.2.2 Range-queries

In this thesis read operations were not considered. It would be interesting to test the performance and efficiency of the proof-of-concept tree with regards to point- and range-queries. This would also cover the difference in number of disk accesses between the Z-Order curve and the Hilbert Curve during queries. And determine if the extra write throughput the Z-Order curve provides is worth it with the reduced query performance compared to the Hilbert Curve.

6.2.3 Bloom filters and Fence pointers

Finding methods to utilize Bloom filters, Fence pointers or similar auxiliary structures for a R-tree could be helpful for reducing the total number of components in the LSM-tree to search. With a LSM-tree with many components, it becomes expensive to do queries if all components has to be searched and read from disk. Filters could then have a huge impact on read throughput.

Bibliography

[1] Patrick O'Neil et al. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385. ISSN: 1432-0525. DOI: 10.1007/s002360050048. URL: https://doi.org/10.1007/s002360050048.

[2] Sattam Alsubaiee et al. "Storage management in AsterixDB". In: *Proceedings of the VLDB Endowment* 7.10 (2014), pp. 841–852.

[3] Sanjay Ghemawat, Jeff Dean, and Google Inc. *LevelDB: A fast and lightweight key/value database library*. URL: https://github.com/google/leveldb.

[4] Facebook. *RocksDB*. URL: https://rocksdb.org/.

[5] Michael Bader. "Two Motivating Examples: Sequential Orders on Quadtrees and Multidimensional Data Structures". In: *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–12. ISBN: 978-3-642-31046-1. DOI: 10.1007/978-3-642-31046-1_1. URL: https://doi.org/10.1007/978-3-642-31046-1_1.

[6] Antonin Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching". In: *SIGMOD Rec.* 14.2 (June 1984), pp. 47–57. ISSN: 0163-5808. DOI: 10.1145/971697.602266. URL: http://doi.acm.org/10.1145/971697.602266.

[7] Yannis Manolopoulos et al. "Introduction". In: *R-Trees: Theory and Applications*. London: Springer London, 2006, pp. 3–13. ISBN: 978-1-84628-293-5. DOI: 10.1007/978-1-84628-293-5_1. URL: https://doi.org/10.1007/978-1-84628-293-5_1.

[8] David Hilbert. "Über die stetige Abbildung einer Linie auf ein Flächenstück". In: *Dritter Band: Analysis · Grundlagen der Mathematik · Physik Verschiedenes: Nebst Einer Lebensgeschichte*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1935, pp. 1–2. ISBN: 978-3-662-38452-7. DOI: 10.1007/978-3-662-38452-7_1. URL: https://doi.org/10.1007/978-3-662-38452-7_1.

[9] George Morton. “A computer oriented geodetic data base and a new technique in file sequencing”. In: 1966.

[10] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie. “The PH-tree”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD ’14* (2014). DOI: 10.1145/2588555.2588564. URL: http://dx.doi.org/10.1145/2588555.2588564.

[11] 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. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 500–509. ISBN: 1-55860-153-8. URL: http://dl.acm.org/citation.cfm?id=645920.673001.

[12] Norbert Beckmann et al. “The R*-tree: an efficient and robust access method for points and rectangles”. In: *Acm Sigmod Record*. Vol. 19. 2. Acm. 1990, pp. 322–331.

[13] Michael Bader. *Space-filling Curves. An Introduction With Applications in Scientific Computing*. Vol. 9. Jan. 2013. DOI: 10.1007/978-3-642-31046-1.

[14] G. Peano. “Sur une courbe, qui remplit toute une aire plane”. In: *Mathematische Annalen* 36.1 (Mar. 1890), pp. 157–160. ISSN: 1432-1807. DOI: 10.1007/BF01199438. URL: https://doi.org/10.1007/BF01199438.

[15] Mohamed F. Mokbel and Walid G. Aref. “Space-Filling Curves”. In: *Encyclopedia of GIS*. Ed. by Shashi Shekhar and Hui Xiong. Boston, MA: Springer US, 2008, pp. 1068–1072. ISBN: 978-0-387-35973-1. DOI: 10.1007/978-0-387-35973-1_1233. URL: https://doi.org/10.1007/978-0-387-35973-1_1233.

[16] Charu C. Aggarwal. “On K-anonymity and the Curse of Dimensionality”. In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB ’05. VLDB Endowment, 2005, pp. 901–909. ISBN: 1-59593-154-6. URL: http://dl.acm.org/citation.cfm?id=1083592.1083696.

[17] R. Borrell et al. “Parallel mesh partitioning based on space filling curves”. In: *Computers & Fluids* 173 (2018), pp. 264 –272. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2018.01.040. URL: http://www.sciencedirect.com/science/article/pii/S0045793018300446.

[18] Takanori Koga and Noriaki Suetake. “Image coarsening by using space-filling curve for decomposition-based image enhancement”. In: *Journal of Visual Communication and Image Representation* 24.7 (2013), pp. 806 –818. ISSN: 1047-3203. DOI: https://doi.org/10.1016/j.jvcir.2013.05.008. URL: http://www.sciencedirect.com/science/article/pii/S1047320313000941.

[19] Yuefeng Zhang and Robert E. Webber. “Space Diffusion: An Improved Parallel Halftoning Technique Using Space-filling Curves”. In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’93. Anaheim, CA: ACM, 1993, pp. 305–312. ISBN: 0-89791-601-8. DOI: 10.1145/166117.166156. URL: http://doi.acm.org/10.1145/166117.166156.

[20] John Mellor-Crummey, David Whalley, and Ken Kennedy. “Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings”. In: *International Journal of Parallel Programming* 29.3 (2001), pp. 217–247. ISSN: 1573-7640. DOI: 10.1023/A:1011119519789. URL: https://doi.org/10.1023/A:1011119519789.

[21] M. F. Mokbel et al. “Scalable multimedia disk scheduling”. In: *Proceedings. 20th International Conference on Data Engineering*. 2004, pp. 498–509. DOI: 10.1109/ICDE.2004.1320022.

[22] R. Kriemann. “Parallel -Matrix Arithmetics on Shared Memory Systems”. In: *Computing* 74.3 (May 2005), pp. 273–297. ISSN: 1436-5057. DOI: 10.1007/s00607-004-0102-2. URL: https://doi.org/10.1007/s00607-004-0102-2.

[23] A. C. Calder et al. “High-Performance Reactive Fluid Flow Simulations Using Adaptive Mesh Refinement on Thousands of Processors”. In: *SC ’00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. Nov. 2000, pp. 56–56. DOI: 10.1109/SC.2000.10010.

[24] B. Moon et al. “Analysis of the clustering properties of the Hilbert space-filling curve”. In: *IEEE Transactions on Knowledge and Data Engineering* 13.1 (2001), pp. 124–141. ISSN: 1041-4347. DOI: 10.1109/69.908985.

[25] C. Faloutsos and S. Roseman. “Fractals for Secondary Key Retrieval”. In: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS ’89. Philadelphia, Pennsylvania, USA: ACM, 1989, pp. 247–252. ISBN: 0-89791-308-6. DOI: 10.1145/73721.73746. URL: http://doi.acm.org/10.1145/73721.73746.

[26] Hans Sagan. “A three-dimensional Hilbert curve”. In: *International Journal of Mathematical Education in Science and Technology* 24 (July 1993), pp. 541–545. DOI: 10.1080/0020739930240405.

[27] H. V. Jagadish. “Linear Clustering of Objects with Multiple Attributes”. In: *SIGMOD Rec.* 19.2 (May 1990), pp. 332–342. ISSN: 0163-5808. DOI: 10.1145/93605.98742. URL: http://doi.acm.org/10.1145/93605.98742.

[28] Arthur R. Butz. “Convergence with Hilbert’s space filling curve”. In: *Journal of Computer and System Sciences* 3.2 (1969), pp. 128 –146. ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(69)80010-3. URL: http://www.sciencedirect.com/science/article/pii/S0022000069800103.

[29] Christos Faloutsos and Yi Rong. “DOT: A Spatial Access Method Using Fractals”. In: (Oct. 1999).

[30] Avinash Lakshman, Prashant Malik, and Apache Software Foundation. *Apache Cassandra*. URL: https://cassandra.apache.org/.

[31] URL: https://en.wikipedia.org/wiki/Hilbert_R-tree.

[32] Ibrahim Kamel and Christos Faloutsos. “On packing R-trees”. In: *Proceedings of the second international conference on Information and knowledge management - CIKM ’93* (1993). DOI: 10.1145/170088.170403. URL: http://dx.doi.org/10.1145/170088.170403.

[33] *AsterixDB*. URL: http://asterixdb.ics.uci.edu/.

[34] Young-Seok Kim. “Transactional and Spatial Query Processing in the Big Data Era”. In: 2016.

[35] Niv Dayan and Stratos Idreos. “Dostoevsky”. In: *Proceedings of the 2018 International Conference on Management of Data - SIGMOD ’18* (2018). DOI: 10.1145/3183713.3196927. URL: http://dx.doi.org/10.1145/3183713.3196927.

[36] Hans-Peter Kriegel, Erich Schubert, and Arthur Zimek. “The (black) art of runtime evaluation: Are we comparing algorithms or implementations?” In: *Knowledge and Information Systems* 52.2 (2017), pp. 341–378. ISSN: 0219-3116. DOI: 10.1007/s10115-016-1004-2. URL: https://doi.org/10.1007/s10115-016-1004-2.