NTNU

Norwegian University of
Science and Technology

Scalable Region Query Processing in Spatial and Spatiotemporal Databases

Vigleik Lund

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

In recent years, there has been an increased focus on providing support for large spatial and spatiotemporal point data by using locality-preserving linearization techniques to transform multi-dimensional data into one-dimensional data and store the transformed data in scalable key-value NoSQL stores. Spatial and spatiotemporal region queries are then executed by decomposing the query into multiple linear range scans on the linearized system and processing the range scans in a filter-and-refine manner. The process of decomposing a multi-dimensional region query into a number of smaller linear range scans and reduce the number of false hits is essential for achieving efficient query processing in these systems. However, there exists a trade-off between the overhead due to creating additional range scans and the overhead due to removing false positives.

We have, in this thesis, studied various existing methods of decomposing spatial and spatiotemporal region queries into linear range scans. Based on the research, we have proposed a cost-driven approach that decomposes a query according to the data distribution, which is estimated by a histogram. The proposed method has been implemented in a simple database prototype and evaluated with real-world test data, and compared against a popular and more space-driven decomposition approach.

The experiments show that our method is able to outperform the baseline approach for both spatial and spatiotemporal point data with most improvement for the latter. One of the main strengths of the proposed method over the baseline is the capability to create an accurate decomposition with a relatively few range scans when dealing with skewed data. The downside is having to maintain a histogram over the data, which can be prohibitively expensive when dealing with large datasets. However, measures such as sampling were found to reduce this cost significantly with minimal impact on performance.

Sammendrag

I de senere år har det vært et økt fokus på å gi støtte til behandling av store mengder romlig og romlig-temporale punktdata ved å bruke lokalitetsbevarende lineæringsteknikker til å transformere flerdimensjonal data til endimensjonal data og lagre den transformerte dataen i skalerbare og nøkkelbaserte NoSQL databaser. Den lineariserte dataen kan da hentes ut med romlig og romlig-temporale region-spørringer ved å dekomponere spørringene til et sett med lineære intervall-spørringer og prosessere disse intervall-spørringene med filter-og-rense teknikk. Dekomponeringsprosessen som dekomponerer en region-spørring til intervall-spørringer og reduseres antall falske treff er essensiell for å oppnå høy ytelse i disse systemene. Men det eksisterer en avveiing mellom kostnadene knyttet til å generere intervall-spørringer og å fjerne falske treff.

Vi har i denne avhandlingen undersøkt de ulike metodene som brukes til å dekomponere romlig og romlig-temporale region spørringer til lineære intervall-spørringer. Basert på denne undersøkelsen så har vi forslått en kostnadsdreven dekomponeringsmetode som dekomponerer spørringer med hensyn til datadistribusjonen ved å bruke histogrammer. Denne metoden har vi implementert i en enkel databaseprototype og evaluert med reelle testdata, og videre sammenlignet resultatene mot en populær og romlig styrt dekomponeringsmetode.

Eksperimentene våre viser at vår metode yter bedre enn baseline mot både romlig og romlig-temporale punktdata. Vår metode demonstrerte størst ytelsesforbedring mot romlig-temporal punktdata. En av de store fordelene med vår metode er at den er i stand til å generere en nøyaktig dekomponering med forholdsvis få antall intervall-spørringer når dataen er ikke-uniform. Ulempen med vår metode er at vi må vedlikeholde et histogram over dataen. Dette kan være uoverkommelig dyrt når man arbeider med store datamengder. Vi klarte å redusere denne kostnaden betraktelig uten å nevneverdig påvirke ytelsen med tiltak som sampling.

Preface

This master thesis was written during the spring of 2018 at the Norwegian University of Science and Technology (NTNU). It is assumed that the reader has a basic knowledge of database systems, algorithms, and data structures.

I would like to thank my supervisor Professor Svein Erik Bratsberg for the support throughout this period and giving me the flexibility to shape the project as I saw fit. I would also like to thank the contributors to the various libraries that have been used throughout the project.

Contents

Summary i

Sammendrag ii

Preface iii

Contents vi

List of Tables vii

List of Figures ix

1 Introduction and Motivation 1
- 1.1 Problem Description . . . 1
- 1.2 Project Goal . . . 2
- 1.3 Scope . . . 2
- 1.4 Structure . . . 2

2 Background 4
- 2.1 Geometric Data Types . . . 4
  - 2.1.1 Spatial Points . . . 4
  - 2.1.2 Spatiotemporal Points . . . 4
  - 2.1.3 Characteristics of Spatial and Spatiotemporal Point Data . . . 5
  - 2.1.4 Polygons . . . 5
- 2.2 Queries . . . 6
  - 2.2.1 Spatial Region Queries . . . 6
  - 2.2.2 Spatiotemporal Region Queries . . . 6
- 2.3 Index Structures . . . 7
  - 2.3.1 B-tree . . . 7
  - 2.3.2 B+tree . . . 8
  - 2.3.3 R-tree . . . 9
  - 2.3.4 Quadtree . . . 9
  - 2.3.5 Other Approaches . . . 10
- 2.4 Space-Filling Curves . . . 11

2.4.1 Z-order Curve . . . 12
2.4.2 Indexing Using Space-Filling Curves . . . 13
2.4.3 Region Queries and Space-Filling Curves . . . 14
2.5 Related Work . . . 17
2.5.1 GeoMesa . . . 18
2.5.2 MD-HBase . . . 19
2.5.3 Cassandra . . . 20

3 Region Decomposition Method **21**
3.1 Motivation . . . 21
3.2 Requirements . . . 21
3.3 Best-first Decomposition . . . 22

4 Methodology and Implementation **24**
4.1 Database Layer . . . 24
4.1.1 Approach . . . 24
4.1.2 MapDB . . . 24
4.2 Access Structures . . . 25
4.2.1 Z-address Calculation . . . 25
4.2.2 Encoding of Space and Time . . . 29
4.2.3 Indexes . . . 31
4.2.4 Insertion Procedure . . . 32
4.3 Query Handling . . . 33
4.3.1 Decomposition Stage . . . 35
4.3.2 Filter Stage . . . 42
4.3.3 Refinement Stage . . . 42
4.4 Parallelization . . . 43
4.4.1 Multi-threaded Decomposition . . . 43
4.4.2 Multi-threaded Refinement . . . 44

5 Experiments, Results, and Discussions **46**
5.1 Baseline . . . 46
5.2 Dataset . . . 47
5.3 Queries . . . 47
5.3.1 Spatial Region Queries . . . 47
5.3.2 Spatiotemporal Region Queries . . . 48
5.4 System Configuration . . . 48
5.5 Metrics . . . 49
5.6 Insertion Performance . . . 50
5.7 Breadth-first Decomposition . . . 52
5.7.1 Maximum Number of Ranges . . . 52
5.8 Best-first Decomposition . . . 54
5.8.1 Maximum Prefix Length of the Histogram . . . 55
5.8.2 Sampling Size of the Histogram . . . 58
5.8.3 Detailed Performance Comparison to the Baseline . . . 59
5.8.4 Query Size . . . 63
5.8.5 Polygon Complexity . . . 64
5.9 Parallelization . . . 66

5.10 Validation 68

6 Conclusion and Further Work **70**
6.1 Conclusion 70
6.2 Further Work 71

List of Tables

4.1 List of successive bit shifts and bit masks that are used to encode 64-bit 2D and 3D Z-addresses. 29

5.1 System configuration. 48
5.2 The various index sizes. 51
5.3 Parameter values used for the best-first decomposition method. 59
5.4 Parameter values used for the breadth-first decomposition method. 59
5.5 Average speedup by using multiple threads. 68
5.6 The number answer points returned from the best-first decomposition method compared to true number of answer points. 69

List of Figures

2.1 Spatial and spatiotemporal point data. 5
2.2 Spatial and spatiotemporal region queries. 7
2.3 B-tree of order 4. 8
2.4 B+tree of order 4. 9
2.5 Variants of the quadtree. 10
2.6 Different types of space-filling curves. 11
2.7 Various three-dimensional space-filling curves [29]. 12
2.8 The recursive construction principle of the Z-order curve. 13
2.9 The data are partitioned into 6 disjoint Z-regions with the UB-tree. 14
2.10 A UB-tree of order 4. The data within a Z-region is stored in a single leaf node. 14
2.11 A region query and its minimum covering Z-range. 15
2.12 Example of a quadtree-based decomposition. 17

4.1 The specific distances we want to move the different bits in coordinate a_1 28
4.2 First pass is a $2^4 = 16$ distance shift. Only bit b_8 and b_9 in a_1 needs to move this far. The bit mask ensures that rest of the bits in a_1 stays unshifted. . . 28
4.3 Second pass is a $2^3 = 8$ distance shift. Bit b_4, b_5, b_6 and b_7 in a_1 are shifted while rest of the bits stays in their position. 28
4.4 Third pass is a $2^2 = 4$ distance shift. 28
4.5 Fourth and last pass is a $2^1 = 2$ distance shift. All bits in a_1 are now spread out correctly. 28
4.6 A more detailed representation of the index keys used in the prototype. . . 31
4.7 Overview of the insertion procedure. 32
4.8 Overview of the query processing procedure. 34
4.9 Calculating the root node of quadtree 37
4.10 The Z-addresses for a two-dimensional Z-order curve with 3 bits per dimension. 37
4.11 Expanding the example root node into $2^2 = 4$ sub-quadrants. 38
4.12 Calculating the Z-bounds z_{lower} and z_{upper} by setting the bits after the prefix to zero and one, respectively. 38

4.13 The priority queue after the first iteration. The priority value reflects the amount of data contained in the quadrant 39
4.14 The priority queue after the first iteration. The priority value reflects the amount of data contained in the quadrant. 40
4.15 Updating the histogram with an index key. 40
4.16 The complete decomposition of the example query. Notice that the hyper-quadrants z_{prefix} = 0000 and z_{prefix} = 0001 are not expanded since their priority value equals zero. 41
4.17 The number of ranges in the decomposition in figure 4.16 can be reduced from nine ranges to five ranges by merging adjacent ranges. The quality of the decomposition remains the same. 42
4.18 Simple multi-threaded decomposition. 44

5.1 Distribution of data in Beijinig [34] 47
5.2 Two example polygons. 48
5.3 Insertion performance. 51
5.4 The effect of increasing the maximum number of ranges on the average response time with the breadth-first decomposition method. 53
5.5 The effect of increasing the maximum number of ranges on the average false discovery rate with the breadth-first decomposition method. 53
5.6 The effect of the maximum prefix length of the histogram on the average response time. 56
5.7 The effect of the maximum prefix length of the histogram on the average false discovery rate. 56
5.8 The effect of sampling size on the average response time. 58
5.9 The effect of sampling size on the average false discovery rate. 59
5.10 The performance of the breadth-first decomposition method and the best-first decomposition method. 60
5.11 Average false discovery rate versus number of ranges created by the best-first decomposition method and the breadth-first decomposition method. 62
5.12 The relationship between the size of the query region and the result set size. 63
5.13 The relationship between the result size and the response time. 63
5.14 The relationship between the measured polygon complexity and the false discovery rate. 65
5.15 The response time of the best-first decomposition method with increasing number of threads. 67
5.16 The decomposition time of best-first decomposition method with increasing number of threads. 67

Chapter 1

Introduction and Motivation

The rate of spatiotemporal and spatial data generated around the globe has increased immensely during the past decade due to advancements in hardware and the expanded use of GPS-enabled smartphones [37]. This has resulted in an increasing number of location-based services that rely on user-generated location data, such as Google Real-time Traffic, which monitors the current traffic situations [6], and Snapchat, which provides location-sharing of users [14].

This growth is expected to continue, especially as the Internet of Things era arrives, which is introducing a need for new and efficient database methods for handling vast quantities of spatial and spatiotemporal data.

1.1 Problem Description

Traditional relational database management systems (RDBMS) handle spatial or spatiotemporal data by using some variation of the R-tree index. The capabilities of an RDBMS to handle an enormous amount of data found in spatial and spatiotemporal applications remain limited, however, as these systems resort to vertical scaling. Horizontal scaling, on the other hand, is necessary to keep up with the growth of data without overwhelming the system. Distributed NoSQL databases emerged in response to the need for scalable databases [22]. However, R-trees and its variants tend to not as efficient in a distributed context as they must be centrally maintained and degrade as the structure grows. The need for rebalancing R-tree index structures can also be prohibitively expensive in many applications.

An alternative approach is to use space-filling curves to linearize multi-dimensional data into the one-dimensional key-space of key-value NoSQL stores. Linearization techniques with locality-preserving properties are used to map spatial and spatiotemporal data directly into one-dimensional cells without information from prior inserted records. This space-driven indexing technique ensures the insertion performance remains constant as more data is inserted and eliminates the need for index rebalanc-

ing. The spatial and spatiotemporal index can then scale with the underlying NoSQL store. Examples of recently developed systems leveraging space-filling curves to support spatial and spatiotemporal data in scalable NoSQL systems include GeoMesa [7] and MD-HBase [31]. These systems, however, present other unique challenges. For instance, the strategy used to process and transform multi-dimensional queries into linear queries supported by the underlying one-dimensional index of the NoSQL store can significantly impact the query performance.

1.2 Project Goal

The goal of this project is to improve the performance of spatial and spatiotemporal query processing in one-dimensional databases. An experimental prototype will be built and evaluated with real-world data, which will be capable of indexing and retrieving spatial and spatiotemporal point data and will implement a proposed method of decomposing multi-dimensional region queries into sets of linear range queries. The proposed method attempts to address limitations found in similar systems with the expectation to develop an improved method of decomposing spatial and spatiotemporal region queries for implementation into existing database systems.

1.3 Scope

The following restrictions on the scope of this project are in place as our time is limited to complete this project.

- The implemented prototype will only deal with spatial and spatiotemporal point data. It will not store spatial objects, such as polygons and linestrings.
- The prototype will only support the retrieval of indexed point data with spatial and spatiotemporal region queries. Features such as visualization or analysis of retrieved data are out of scope.
- The prototype implemented will be an embedded database system without support for distributed indexing and processing.
- The focus of this project is not to develop a database system that can directly compete with existing systems, but to demonstrate a proposed method.

1.4 Structure

The rest of the project is structured in the following manner:

- **Chapter 2** provides the theoretical background and discusses some of the related work.
- **Chapter 3** describes our proposed method for optimizing the performance of region queries.
- **Chapter 4** gives the implementation details of the prototype.

- **Chapter 5** presents the experiments, results and the discussions.
- **Chapter 6** sums up the work and presents suggestions for further work.

Chapter 2

Background

This chapter provides the theoretical background for the thesis. The first part introduces relevant data types, queries, and index structures. The second part covers space-filling curves and how they can be employed to retrieve and manage spatial and spatiotemporal point data. The final section discusses related spatial and spatiotemporal NoSQL systems that rely on space-filling curves to handle large datasets.

2.1 Geometric Data Types

Geometric data types found in spatial and spatiotemporal databases can be divided into two categories of raster and vector data. Raster data represents features with a regular grid of pixels where each pixel has an associated value. Vector data represent points and polygons as discrete features. This project deals with vector data where objects are stored as spatial and spatiotemporal point data, and polygons define areas of interests in our queries.

2.1.1 Spatial Points

Spatial point data is multi-dimensional data that represent points defined in a geometric space such that the location of a point is represented by two geographical coordinates (latitude and longitude). Spatial point data may include additional attributes providing information about the represented entities. Spatial points can be recorded in the following record format:

$$P_{spatial} = (Longitude, Latitude, Identifier, Attr_1, Attr_2, .. , Attr_n)$$

2.1.2 Spatiotemporal Points

Spatiotemporal point data is spatial point data with an additional temporal component used to describe the evolution of an entity's position over time. When an entity changes location, its new recorded position in space creates a new snapshot that includes a timestamp. Spatiotemporal points can be recorded in the following record

format:

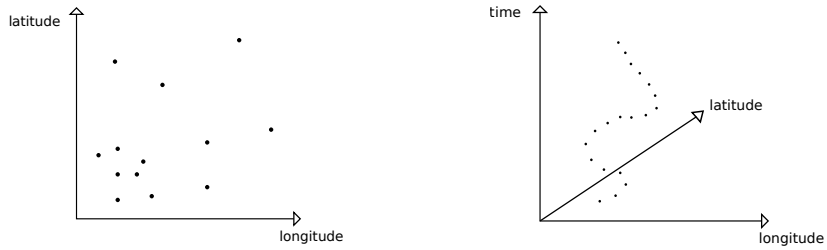
$$P_{spatiotemporal} = (Timestamp, Longitude, Latitude, Identifier, Attr_1, Attr_2, .. , Attr_n)$$

2.1.3 Characteristics of Spatial and Spatiotemporal Point Data

Several important characteristics of spatial and spatiotemporal point data must be considered when designing spatial and spatiotemporal indexing structures:

- Both the temporal and spatial components are inherently skewed. For instance, certain places at specific times, such as cities at daytime, will be denser compared to rural regions at nighttime.
- The spatial component has no a priori ordering required for lexicographical ordering.
- The temporal component is potentially unbounded.

An additional important note when dealing with spatiotemporal point data is that any assumption about the movement between the snapshots may lead to incorrect information. It is common to assume an entity remains stationary at the position given its last stored snapshot. If we use linear interpolation for building entity trajectories, then movements along these trajectory segments may falsely suggest an entity has entered a restricted area when it has moved around it.



(a) Example of spatial point data.

(b) Example of spatiotemporal point data.

Figure 2.1: Spatial and spatiotemporal point data.

2.1.4 Polygons

A polygon is a spatial region bounded by line segments, and is defined with an ordered set of spatial points representing the vertices of the polygon:

$$R = (Longitude_1, Latitude_1, Longitude_2, Latitude_2, ..., Longitude_n, Latitude_n)$$

The consecutive pairs of points define the line segments that bounds the interior of the polygon. A polygon must have at least three vertices, and the line segments joining two vertices must not intersect (except for *complex polygons*, which are used in this thesis).

2.2 Queries

Indexing and storing spatial and spatiotemporal point data enable responding to location and time-based queries. Typical query types used in spatial and spatiotemporal point databases include the nearest neighbour, point, and region queries. In this thesis, we focus the latter. Region queries are a traditional problem in spatial and spatiotemporal databases and are useful in the domain of analyzing spatial and spatiotemporal data over areas and time as well as for generating maps and models.

2.2.1 Spatial Region Queries

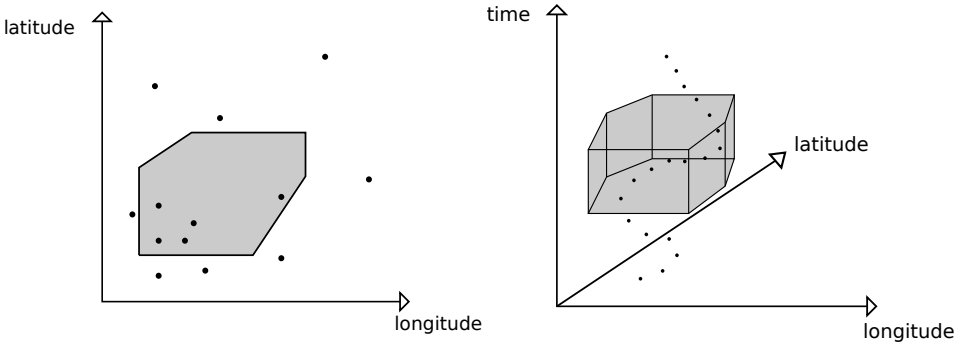
A spatial region query is specified by a polygonal region R. The output is the set of all points P that intersect with region R. A two-dimensional visualization of a spatial region query is shown in Figure 2.2a. The range query is a special case of the region query. The region R in a spatial range query is specified as an interval over the spatial dimensions, i.e., a box-shaped region defined by its lower-left and upper-right corners.

An example of a relevant spatial region query is *"find all entities within Trondheim City."* The query would return a list of all the stored entities contained by the polygon of Trondheim City.

2.2.2 Spatiotemporal Region Queries

A spatiotemporal region query is an extension of the spatial region query with a temporal component. In addition to a spatial query region R, a time range T is included. The output of a spatiotemporal region query is all the points that intersect with the query region R and the time interval T. A spatiotemporal region query can be visualized as a polyhedral in three-dimensional space, as shown in Figure 2.2b.

An example of a relevant spatiotemporal region query is *"find all entities within Trondheim City between 2/5/2016 5:00:00 and 4/6/2016 5:00:00."* This query would return a list of all the entities having an indexed snapshot with spatial coordinates and a timestamp that intersect with the spatial region of Trondheim City and the time range 2/5/2016 5:00:00–4/6/2016 5:00:00.



(a) Example of a spatial region query.

(b) Example of a spatiotemporal region query.

Figure 2.2: Spatial and spatiotemporal region queries.

2.3 Index Structures

Indexing enables efficient retrieval of data at the cost of building and maintaining an index structure. Without indexes, we would have to perform complete table scans to retrieve the desired data. Full table scans are prohibitively expensive in most applications, especially when dealing with extensive data sets found in spatial and spatiotemporal databases.

2.3.1 B-tree

The B-tree is a self-organizing and high-performing index commonly found in database systems. The B-tree index structure stores the data sorted and provides a logarithmic performance guarantee for all basic operations, such as insert, delete, and lookup. This allows the structure to handle large data sets as offers fast retrieval and insertion speed independent of the data set size. Additionally, very little main memory is required to cache the upper levels of the tree or a search path to a leaf node. Each insertion or delete operation in the B-tree results in a reorganization of the tree structure. This self-organization property enables continuous operation without interruptions for periodical reorganizations, and makes the index robust against highly dynamic data.

Each node in the B-tree index may contain more than two keys. Internal nodes with K keys has $K+1$ child nodes. The keys within a node are stored in sorted order with alternating pointers to its child nodes: $pointer_1, key_1, pointer_2, key_2$, and $pointer_3$. The child represented by $pointer_1$ contains key values less than key_1. The child node of $pointer_2$ contains keys between key_1 and key_2, and the last child of $pointer_3$ contains keys greater than key_2. Data is retrieved by performing a top-to-bottom search until the desired key and the associated value is found. At each level, the child node with a key range that includes the search key is accessed. For a B-tree with a height of five levels, the worst-case scenario for a lookup operation accesses five nodes, although disk accesses can be reduced by caching the upper levels of the B-tree. In practical situations, B-trees typically provide a guaranteed access time of less than 10 ms for extremely large datasets [2].

The key idea to understanding the B-tree structure is the split procedure that reorganizes the tree during insertion. The interior nodes, except the root, of a B-tree of order m has at least $\lceil \frac{m}{2} \rceil$ children and $\lceil \frac{m}{2} \rceil - 1$ keys. Also, each node has at most m children and $m-1$ keys. When trying to insert an object into a full node, the node is split into two half-full nodes with $\frac{m}{2}$ elements. The middle of the two new nodes is inserted into their common parent node in sorted order, and the splitting can propagate recursively upwards to the root node. The depth of the tree will only increase when splitting the root node, which results in the creation of a new root node with a single middle element and pointers to its two child nodes. The number of split operations performed during insertion is bound by the height of the tree, which gives the insertion operation a logarithmic complexity.

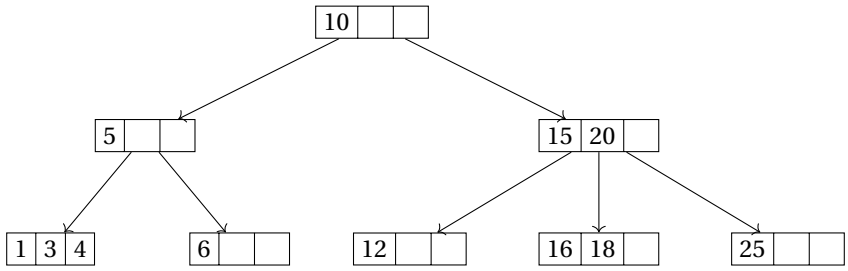


Figure 2.3: B-tree of order 4.

2.3.2 B+tree

A variant of the B-tree is the B+tree with its main difference being that the interior nodes in the B+tree contain only keys and not values, and data is only associated with the leaf nodes. Additionally, the leaf nodes in a B+tree are linked. This makes sequential access very efficient in B+trees compared to B-trees, which requires a separate traversal from root to each leaf node. A range scan in a B+tree is performed with a single traversal from root to leaf node followed with a linear pass through the linked leaf nodes. Otherwise, the behaviour of the B+tree is mostly the same as the B-tree, except during the splitting of a leaf node, where the middle element remains in the right child node, and only the key (not the value) is transferred to the parent node. Fewer data in the intermediate interior nodes allow for a higher branching factor and a potentially shallower tree. The large fanout usually outweighs the disadvantage of not being able to access data in internal nodes directly as most of the accessed data are found at the bottom of the tree.

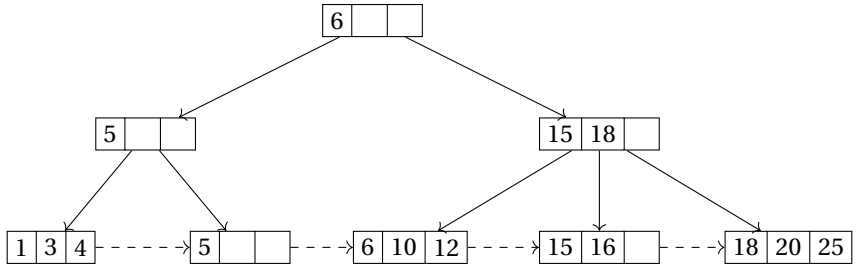


Figure 2.4: B+tree of order 4.

2.3.3 R-tree

The R-tree index is a spatial version of the B+tree index where nodes are represented as minimum bounding rectangles (MBR). R-trees store data in leaf nodes and uses internal nodes to group and enclose leaf nodes (or internal child nodes) that are nearby spatially with MBRs. The data become more aggregated by the internal nodes further up in the tree, and the MBR of the root node encloses the entire data set. This can be thought of as an increasingly coarse approximation of the data set. A region search is performed top-down in an R-tree. The idea is to recursively traverse child nodes with an MBR that intersects with the query region. Data within the reached leaf nodes are validated against the exact query geometry and returned if found to be fully contained by the query region.

The main difficulty with the R-tree is the construction of an efficient tree structure that provides good worst-case performance, especially when dealing with dynamic data sets with random inserts [13]. Contrary to the B-tree index, the R-tree does not come with logarithmic worst-case performance guarantees for the basic operations. To obtain the best performance, the tree should be balanced and contain MBRs that neither overlap too much nor contain too much empty space, which restricts the number of subtrees to be searched. A popular variant of the R-tree that tries to reduce MBR overlap is the R*-tree [19], which provides better search performance at the cost of a significant insertion overhead.

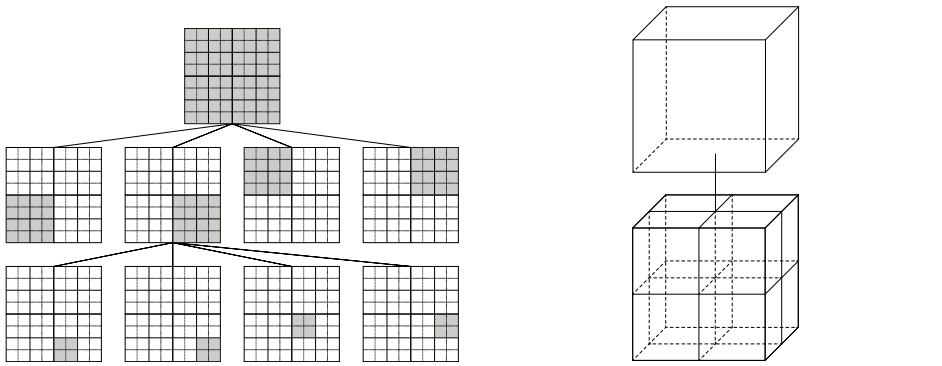
Even though R-trees are used mostly for spatial data, they can be extended to support spatiotemporal data. An example of this scenario addresses time as a third and separate dimension with nesting of the data in minimum bounding cuboids instead of rectangles. Spatiotemporal R-tree structures based on this approach include 3D R-trees, RT-trees, and STR-trees [30].

2.3.4 Quadtree

A quadtree is a tree data structure that recursively decomposes space into subspaces called quadrants. Each internal node in a quadtree has four children, and the child nodes partition the space enclosed by the parent. The root node covers the entire space, and the leaf nodes contain the indexed points. There exist various types of

quadtrees, which are categorized into trie-based or point-based approaches. The trie-based quadtree partitions space into equal-sized quadrants while point-based quadtrees adjust the quadrant sizes to the data distribution. This thesis deals with trie-based quadtrees. Spatial point data can be stored in a quadtree by recursively partitioning quadrants until the leaf nodes contain a maximum of one point. The maximum precision of the quadtree is defined by the maximum tree depth or the minimum size of the leaf nodes. Region queries are processed in quadtrees in a similar manner as R-trees. The tree structure is traversed top-to-bottom along the nodes that intersect with the query geometry. Data within reached leaf nodes are returned in the result set.

The N-dimensional analogue (or generalization) of the quadtree is the hyper-quadtree, which divides an N-dimensional hyper-quadrant into 2^N hyper-quadrants.



(a) A trie-based quadtree.

(b) Octree [15]: A three-dimensional equivalent of the quadtree, where space is recursively subdivided into eight octants.

Figure 2.5: Variants of the quadtree.

2.3.5 Other Approaches

Scaling is a challenge with traditional spatial and spatiotemporal indexes based on quadtrees and R-trees used in relational databases. The performance of these structures tends to degrade as the index grows and needs periodic re-balancing. They are also difficult to build and maintain as distributed indexes. In response to the scaling issues related to the growth in spatial and spatiotemporal data, a significant effort has been devoted in recent years to implement spatial and spatiotemporal support in BigTable-style NoSQL systems [23].

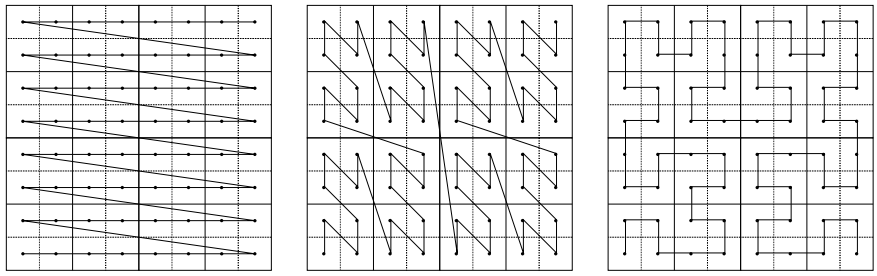
NoSQL systems are proven to handle extremely large dataset efficiently [24]. However, these systems are often restricted to a simple data model with data access over primary key, which makes multi-attribute access impossible without full scans. The idea has been to use locality preserving linearization techniques that map multi-dimensional attributes into one-dimensional space. Keeping a total ordering of points in space while preserving locality makes it possible to support efficient multi-dimensional query pro-

cessing with sorted one-dimensional indexes used in key-value NoSQL stores. The use of space-filling curves is a popular approach to linearize multi-dimensional data.

2.4 Space-Filling Curves

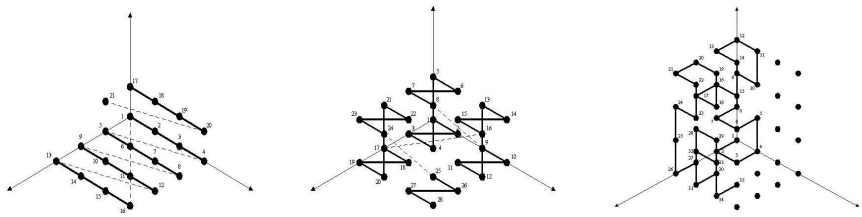
A space-filling curve (SFC) is a locality preserving method of mapping N-dimensional data to a one-dimensional space. The mapping can be visualized as a continuous line passing through every cell in a high-dimensional space exactly once, which imposes a linear ordering of the cells it visits. The quality of the linear ordering given a space-filling curve is defined by its ability to preserve locality. In other words, points close in the initial high-dimensional space should also be close in the mapped one-dimensional space. This property is essential for efficient retrieval of multi-dimensional data in SFC indexes.

There exist various types of space-filling curves, and the differences are in how the curve traverses the high-dimensional space. The different mapping schemes can be categorised into recursive space-filling curves (RSFC) and non-recursive space-filling curves [28]. RSFCs are based on decomposing the space recursively into four equal-sized fragments to a preferred level of precision and traversing the fragments in a pre-determined order. A fragment is traversed one-at-a-time exhaustively at any level of refinement. Two well-known recursive space-filling curves are the Z-order curve (also called Morton order) and the Hilbert curve. Non-RSFCs also traverse the space in a pre-determined order but are not fractal-based and more straightforward to implement. An example is the Sweep curve, which is a one-way traversal of space, that traverses a two-dimensional space horizontally bottom-up and one row at a time. Sweep curves are, however, not optimal in terms of locality-preserving mapping [28].



(a) Sweep curve **(b)** Z-order curve **(c)** Hilbert curve

Figure 2.6: Different types of space-filling curves.



(a) Sweep curve **(b)** Z-order curve **(c)** Hilbert curve

Figure 2.7: Various three-dimensional space-filling curves [29].

2.4.1 Z-order Curve

The Z-order curve, which also goes by the names Peano curve or Morton curve, is a recursive construction with locality-preserving properties and is one of the most commonly used SFC along with the Hilbert curve. The Z-order curve is one of the simplest RSFCs as the encoding procedure consists of bit-interleaving a point's coordinate values to obtain its position on the curve.

A simple two-dimensional example can demonstrate the concept of the Z-order curve. The space in Figure 2.8a is first split into four quadrants. The integer values defining the position along the axis is converted into a binary representation. Each quadrant is then represented by doing a pairwise interleaving of its binary-represented coordinate values. The x-coordinate gives the most significant digit of the quadrant, and the y-coordinate gives the least significant digit. The number of quadrants depends on the number of bits representing the possible coordinate values and defines the resolution of the mapping scheme. If the number of bits used to represent the possible coordinates is incremented by one, then each of four existing quadrants is partitioned into four smaller, equal-sized quadrants, as seen in Figure 2.8b. When a smaller quadrant is contained within a coarser quadrant, then the coarser quadrant is a prefix of the smaller quadrant. Common prefixes imply closeness between quadrants even though there are exceptions, which can be seen by the occurrences of long diagonal jumps. These jumps, which are absent in the more computationally-expensive Hilbert curve, occurs between consecutive connected points across quadrants and become larger as the curve resolution increases.

The Z-order curve visits the quadrants in lexicographical order according to their binary representation. As shown in Figure 2.8, space is traversed in a 'Z'-like pattern, hence the name Z-order curve. Based on the given example, the Z-values for spatiotemporal point data can be calculated directly by bit-interleaving the binary-representation of the latitude, longitude, and timestamp values. Sorting the Z-values generates a three-dimensional Z-order curve. The bit-interleaving procedure can be generalized trivially to any number of dimensions. Furthermore, a data point is known to be within a given quadrant if the Z-value of the given quadrant a prefix of the data point's Z-value.

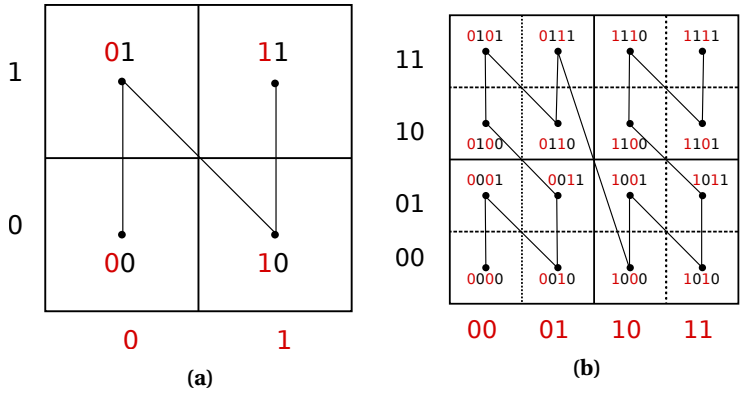


Figure 2.8: The recursive construction principle of the Z-order curve.

2.4.2 Indexing Using Space-Filling Curves

Once the multi-dimensional data has been linearized with a space-filling curve, any one-dimensional index structure can be used to store and retrieve the data. A prominent example is the Universal B-tree (UB-tree) [18], which combines the Z-order curve with the B+tree index.

UB-tree

The UB-tree is an extension of the B+tree for organizing and accessing multi-dimensional point data and leverages the Z-order curve to index multi-dimensional data in a traditional B+tree. The idea is to reduce the problem of searching multi-dimensional data to a problem of searching data in linearly ordered sets using Z-values as keys. The UB-tree retains the logarithmic performance guarantees of the B+tree for all basic operations of insertions, deletion, and lookup. This also includes a guarantee of 50% page utilization, and the logarithmic performance guarantee makes the structure particularly scalable, robust, and suitable for handling large dynamic datasets.

The tree structure of the UB-tree represents a hierarchical partitioning of the Z-order curve where an inner node encloses the space of its children. Space covered by nodes at the same depth of the tree will not overlap (see Figures 2.9 and 2.10). Guaranteed overlap-free partitioning of the data makes the UB-tree more robust against random insertions compared to the R-tree, which tends to degrade due to increased MBR overlap. A look-up in the UB-tree will always be limited to only one search path. Another important advantage of UB-trees over other traditional multi-dimensional index structures is that it can easily be implemented on top of existing database systems that provide B+tree indexes by adding a simple preprocessing technique.

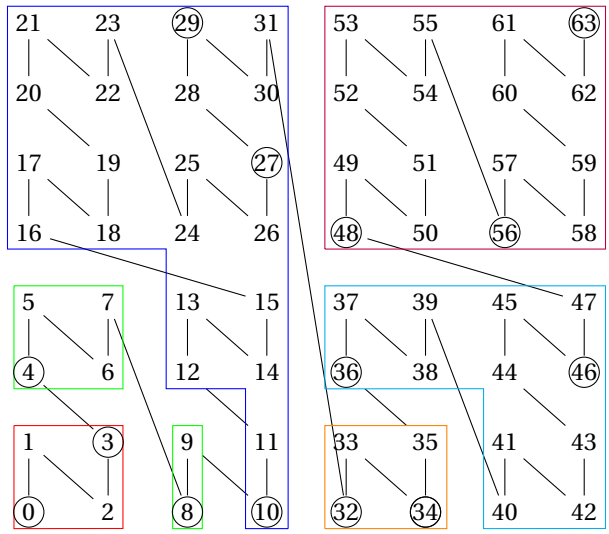


Figure 2.9: The data are partitioned into 6 disjoint Z-regions with the UB-tree.

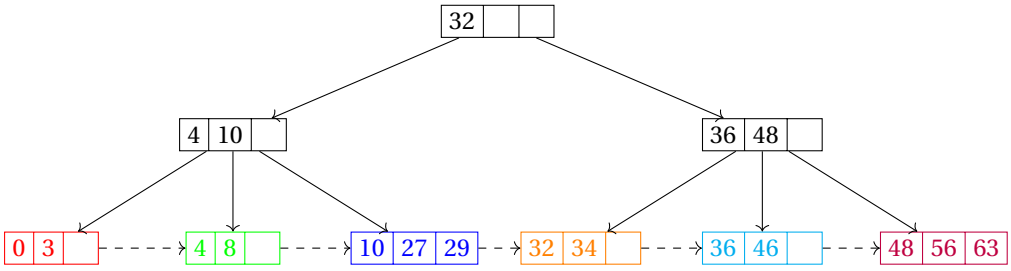


Figure 2.10: A UB-tree of order 4. The data within a Z-region is stored in a single leaf node.

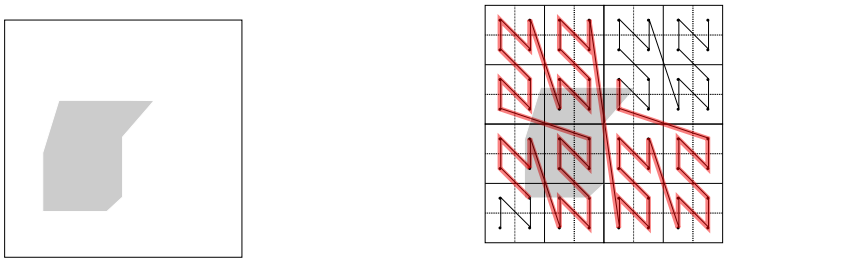
The precision of the Z-order curve used to encode the index keys should be high enough to not cause any overflow by mapping different positioned points to the same key (smallest quadrant at maximum precision). Increasing the precision beyond this point requires, however, more storage and performance overhead.

2.4.3 Region Queries and Space-Filling Curves

Region queries are processed in space-filling curve indexes (like the UB-tree) by using a filter and refine strategy. The initial filter step retrieves the set of candidate points from the one-dimensional index that may overlap with the query region. The filtering is usually constrained to contain false positives. This guarantees no false negatives, as the candidate set will always be a superset of the answer set. The refinement step must inspect the points in the candidate set and remove any potential false positives that are disjoint with the query region. The refinement step is usually the most expensive stage as each inspection involves a point-in-polygon test.

The candidate set is retrieved from the order-preserving linear index by executing one or more range queries where the total space enclosed by the range queries fully encloses the query region. Each range query is specified as a Z-range (interval) on the space-filling curve, where a minimum and maximum Z-value give the Z-range.

The naive approach to executing a region query is to calculate the minimum segment of the curve that thoroughly covers the query region. The lower and upper bounds of the segment are then be executed as a single range query in the filtering step. The issue with this approach is the potential amount of surplus search space with false positives that need to be removed in the expensive refinement step, which is illustrated in Figure 2.11b.



(a) Query region.

(b) The minimum covering Z-range of the query geometry is highlighted in red.

Figure 2.11: A region query and its minimum covering Z-range.

As a single Z-range will rarely provide a good approximation for a query region, a better approach is to decompose the query region into a set of smaller, non-overlapping Z-ranges. Each Z-range from the decomposition can be identified as either contained within or intersecting with the query region. Data retrieved from contained ranges require no validation and can be added to the result set directly. Points from intersecting ranges might be false positives and must be refined. This process is depicted in Algorithm 1.

Algorithm 1 Region query

```
1:  /* Q_R be the query region. */
2:  ℤ ← regionDecomposition(Q_R) /* Partition Q_R region into set of Z-ranges. */
3:  ℝ ← ∅ /* Initialize an empty set to store our result points*/
4:  for each range z ∈ ℤ do
5:      if z ⊆ Q_R then
6:          ℝ ∪ (p ∈ s)
7:      else if z ∩ Q_R then
8:          for each point p ∈ s do
9:              if p ⊆ Q_R then
10:                 ℝ ∪ p
11: return ℝ
```

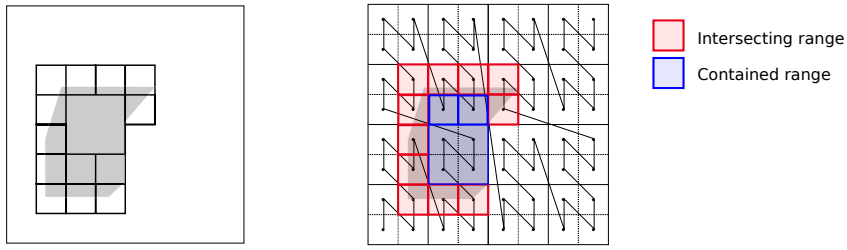
The procedure of decomposing the query region into a series range can be more intricate. A possible approach is to perform a quadtree-based decomposition of the query region [33]. The partitioning scheme of the trie-based quadtree shares significant similarities with the partitioning scheme found in the Z-order curve. In fact, a quadtree can be represented as a linear array of Z-values (of various precision) where each Z-value in the array corresponds to a leaf node of the quadtree, and this structure is called a linear quadtree [17]. Scanning the linear quadtree in sorted order will traverse the leaf nodes of the quadtree in a depth-first manner. The path from the root node to the leaf node is implicitly preserved in the Z-values as they are calculated by appending a sequence of directional bits. For example, a leaf node with a Z-value of $x_1 y_1 x_2 y_2 x_3 y_3 x_4 y_4$ and $x_i, y_i \in \{0,1\},\ \forall i \in \mathbb{Z}$ will have the following path in the quadtree:

$$entire\ space \rightarrow x_1 y_1 \rightarrow x_1 y_1 x_2 y_2 \rightarrow x_1 y_1 x_2 y_2 x_3 y_3 \rightarrow x_1 y_1 x_2 y_2 x_3 y_3 x_4 y_4$$

The interrelationship between the quadtree and the Z-order curve can be leveraged by decomposing the query region into a linear quadtree structure. This is performed by recursively subdividing the space that covers the query region into a set of disjoint (axis-aligned) quadrants. Space is recursively partitioned into quadrants in a breadth-first manner, and the Z-value representation of a quadrant can be mapped directly to a Z-range. Quadrants that are disjoint with the query region are pruned during the decomposition process. The decomposition stops once all remaining quadrants are fully contained by the query region or the cell level (maximum curve precision) is reached.

There is a trade-off between the number of ranges generated and the accuracy of the region represented by the ranges. More ranges introduce additional overhead but also fewer false positives. This can be partially solved by either limiting the maximum precision of the quadtree or the number of ranges created.

The quadtree-based breadth-first decomposition is depicted in Algorithm 2, which works as follows. First, starting with the root node of the quadtree, check if the area covered by the node is either (1) disjoint with the query region, (2) completely covered within the query region or (3) intersects with the query region. In the first case, the node is pruned. In the second case, the Z-range of the node is added to the list of ranges scans. In the third case, the node is expanded, and all its children are rechecked as in the first step. The splitting process continues recursively until some maximum depth is reached or a maximum number of ranges are generated. An example of a decomposition using this method is shown in Figure 2.12b. Various strategies can be used to reduce the size of the generated set of ranges further. Two possible strategies include merging neighbouring ranges and merging ranges that are close together in space. The quadtree-based decomposition method can be trivially extended to any number of dimensions with a hyper-quadtree, e.g., by using a three-dimensional hyper-quadtree (octree) for the spatiotemporal data.



(a) Quadtree representation of the query geometry.

(b) Set of Z-ranges from the quadtree decomposition.

Figure 2.12: Example of a quadtree-based decomposition.

Algorithm 2 Quadtree-based breadth-first decomposition of a query region

```
1:  /* Q_R be the query region. */
2:  𝕊 ← ∅ /* Initialize an empty set to store result ranges.*/
3:  R ← ∅ /* Initialize an empty set to store remaining quadrants to process. */
4:  level ← 0 /* Level of recursion. */
5:  q_r ← root(Q_R) /* Root quadrant enclosing whole search space. */
6:  if q_r ⊆ Q_R then return zOrderRange(q_r)
7:  ENQUEUE(R, q_r) */ Queue up the root quadrant for processing. */
8:  while level < maximumDepth and R ≠ ∅ and |R|+|𝕊| < maximumRanges do
9:      if level is fully processed and  then
10:         level ← level + 1
11:     else
12:         q ← DEQUEUE(R)
13:         for each subquadrant q_s in q do
14:             if q_s ⊆ Q_R then
15:                 𝕊 ∪ zOrderRange(q_s)
16:             else if q_s ∩ Q_R then
17:                 ENQUEUE(R, q_s)
    /* Add partially overlapping ranges that did not get fully processed because the recursion limit was hit. */
18: while R ≠ ∅ do
19:     q ← DEQUEUE(R)
20:     𝕊 ∪ zOrderRange(q)
21: return 𝕊
```

2.5 Related Work

The following section presents and discusses some of the recent approaches of using space-filling curves to support spatial and spatiotemporal point data with NoSQL systems. The primary focus is on the different query processing strategies used to support region queries.

2.5.1 GeoMesa

GeoMesa [7] is an Apache-licensed open-source project created by CCRi that provides indexes for large spatial and spatiotemporal data on top of BigTable-style NoSQL key-value stores, such as Accumulo. GeoMesa is under active development and, according to its authors, can store petabytes of spatial and spatiotemporal data to serve millions of points in seconds with horizontal scaling [8]. GeoMesa utilizes a Z-order curve to linearize spatial and spatiotemporal point data. The data are stored sorted across multiple servers with Z-values as primary keys. Within a server, the key-value pairs are typically indexed with a B-tree style index [1].

Region queries are executed in GeoMesa by partitioning the queries into a series of interval queries or range scans by using a quadtree-based, breadth-first decomposition strategy. GeoMesa decomposes the minimum bounding rectangle of the query region instead of the region itself. The hyper-quadtree is searched recursively for Z-ranges covering the MBR of the query region until either a maximum number of ranges or a recursive limit is reached. Decomposing the MBR makes the process of creating ranges cheaper as the intersection and containment tests of hyper-quadrants consist of rectangle-in-rectangle tests instead of the more complicated rectangle-in-polygon tests. The goal of the decomposition is to reduce false positives due to poor locality preservation in the Z-order curve, which can be seen in the naive minimum covering Z-range example described in the previous section (Figure 2.11b). The breadth-first decomposition does not consider skewness in the data, and the maximum number of ranges for spatial queries is a pre-defined property value that must be tuned for the cluster and the expected query pattern and data sets.

A drawback with the approach of decomposing and filtering the minimum bounding rectangle of the query region is the number of false positives to be scanned. This applies particularly to irregularly shaped query regions as the space difference between query region and its minimum bounding rectangle box is likely to contain a significant amount of data. GeoMesa does not make any distinction between contained and intersecting ranges. Instead, they rely on push-down predicates to filter and refine all ranges in a distributed manner. The ranges in Accumulo, for example, are processed in parallel across the cluster by initiating a `BatchScanner` job. Even though GeoMesa mitigates the cost of removing false positives to some degree by utilizing parallelism during refinement, the average throughput is shown to degrade significantly with query regions of high complexity compared to box-shaped queries [25].

Partitioning the query regions into intersecting and contained ranges would likely reduce the overall stress on the cluster and throughput of high volume and complex-shaped queries. The cost, however, will be a slower execution time of low volume queries due to increased overhead from a more complex decomposition procedure and the need to execute two different `BatchScanner` jobs. Experiments performed with GeoMesa show that the time spent on decomposition and issuing a query already tends to dominate the overall response time with low volume queries [25], so the potential improvement in performance will depend on the query pattern and the data set.

2.5.2 MD-HBase

MD-HBase [31] is an extended version of the NoSQL key-value HBase that provides scalable performance for storing and querying spatial and spatiotemporal point data. Like GeoMesa, MD-HBase relies on a Z-order curve to linearize multi-dimensional data into a one-dimensional and lexicographical ordered key space. The underlying data storage layer of HBase stores the linearized key-value pairs across multiple servers. Within a server, the data is stored in B+tree indexes. MD-HBase implements an overlying index layer on top of HBase to provide efficient multi-dimensional query processing. The overlaying index layer consists of two possible indexes: a linear trie-based quadtree and a kD-tree. Both indexes are directly coupled with the Z-order curve and are built with the inserted linearized data. Each leaf node in the quadtree and the kD-tree index corresponds to a Z-order range and are stored in separate tables in HBase. A leaf node in the quadtree and the kD-tree index is split when the number of points it covers in space exceeds a specified maximum capacity. Dense regions will be split frequently and covered by smaller and more precise leaf nodes compared to sparse regions, which will be covered by a few large leaf nodes. This allows the overlying indexes to capture the data distribution of the stored data.

The MD-HBase paper does not describe any support for region queries and covers only range queries. The axis-aligned box represented by a query range is decomposed into several Z-ranges. Contrary to GeoMesa, MD-HBase does not use a quadtree-based breadth-first decomposition. Instead, it directly retrieves a candidate set of Z-ranges for the decomposition from the linear quadtree index (or the optional kD-tree index). The decomposition process of MD-HBase starts by calculating the minimum covering Z-range of the query range (see Figure 2.11b), which is used to query the quadtree index structure with a linear range scan. This range query returns the minimum set of non-overlapping leaf nodes in the quadtree index that completely covers the minimum covering Z-range of the query range. Since the Z-order curve preserves locality loosely, portions of the minimum covering Z-range may be disjoint with the query range. This means some of the leaf nodes may also be disjoint with the query range. The next step checks each of the returned leaf nodes and determines by using rectangle-in-rectangle checks if they intersect, are contained within or are disjoint with the query range.

Z-ranges of intersecting and contained leaf nodes are executed in parallel as linear range queries in HBase, where points from intersecting Z-ranges are post-refined, and points from contained Z-ranges are retrieved directly. The set of disjoint Z-ranges are then pruned. The split rule of the overlying quadtree index restricts the amount of data returned from a single Z-range. The result is that the decomposition will adapt to data skewness by creating more precise Z-ranges for dense regions.

Selecting the maximum capacity of the leaf nodes in the overlying index structures is described as a trade-off between insertion and query performance. Lowering the capacity will increase the pruning power, and the number of false positives as the quadtree index and the decomposition will be more fine-grained. Lowering the capacity will also increase the split frequency of the quadtree index, which will affect ingestion performance negatively. Experiments performed with MD-HBase shows that the maintenance of an overlying quadtree index layer reduces the insertion performance signif-

icantly compared to the cost of only linearizing the data. Additionally, the decomposition time tends to dominate the total response time with high selectivity queries. Reducing the maximum capacity of the leaf nodes in the quadtree index too much may also degrade query performance of large-sized queries due to the significant increase in the number of candidates ranges returned from the overlying index.

As mentioned earlier, MD-HBase does not support region queries. The following are two possible ways to extend the approach of using an overlying quadtree index during decomposition.

1. Calculate the minimum covering Z-range of the query region and perform rectangle-in-polygon tests to determine if the candidate Z-ranges from the overlying quadtree index intersects, are disjoint or are fully contained by the query region.

2. Perform breadth-first decomposition of the query region and only expand quadrants with Z-values that match a prefix in the quadtree index.

2.5.3 Cassandra

Brahim, et al.[20] demonstrates a spatial framework for Cassandra, which is a fully-distributed, wide-column NoSQL store that uses hash partitioning on the primary key to distribute the data across servers. The data is sorted lexicographically within a server by one or more clustering keys, also called columns. The framework relies on geohashes to linearize spatial data, which is based on the Z-order curve. A geohash value is a Z-value with arbitrary precision represented as an ASCII string. Calculating the geohash value of a point is performed by bit-interleaving the coordinate value to the desired precision and storing the result with base32 encoding. The geohash is used in the framework as a clustering key.

Region queries are executed by first decomposing the region into a set of geohash ranges. The exact implementation of the method is not given in detail but seems to be based on a static breadth-first-like decomposition of a quadtree. The decomposition starts by computing the largest quadrant, which is represented by a geohash value, that is fully contained by the region. The remaining space of the query region is filled with increasingly smaller quadrants (i.e., increasing geohash precision) until the entire region is covered or a maximum geohash precision is reached. Adjacent and overlapping ranges are then merged before being finally executed as range queries in Cassandra. The paper does not mention a distinction between overlapping and intersecting ranges or the removal of false positives.

Based on the experimental results from the paper, it can be assumed that the method relies on a very precise decomposition of the region to retrieve the desired data and only creates fully contained ranges. A maximum precision parameter restricts the precision of the decomposition. A low precision limit is likely to make the result set of irregular shaped, and small-sized region queries inaccurate. On the other hand, increasing the precision limit makes the performance very sensitive to large, complex shaped regions due to many ranges generated. The sensitivity towards large and complex-shaped regions is demonstrated with the experiments performed in the paper.

Chapter 3

Region Decomposition Method

In this chapter, we propose a method for decomposing a region query into a set of Z-ranges.

3.1 Motivation

As described in the previous chapter, several frameworks have been recently developed for handling a large collection of spatial and spatiotemporal point data in NoSQL systems by using space-filling curves. A key factor in executing spatial and spatiotemporal region queries efficiently in these systems is pruning subspaces and reducing the number of false positives scans. This is achieved by decomposing the region query into a series of smaller linear range queries. However, there is a trade-off between the reduced overhead from filtering less false positives and the overhead due to an increase in range queries. The decomposition method needs to balance the benefit of scanning less false positives and the cost of creating more ranges. The method we propose in this chapter improves some of the identified flaws in these approaches, and the primary goal is to optimize the query performance of region queries against space-filling curve indexes.

3.2 Requirements

It is assumed that the data is indexed by their Z-value and stored lexicographically, and the underlying storage layer supports basic one-dimensional range queries.

Query Types
The method should support spatial and spatiotemporal region queries for point data, the spatial predicate should be a simple polygon (not self-intersecting or with holes) of arbitrary shape, and the temporal predicate should be a time interval.

Performance
Both spatial and spatiotemporal queries should be executed fast and the response time should depend on the amount of data in the result set and not the index size.

Scalability
The method should have minimal effect on the insertion performance of the system, and it should not cause query performance and insertion performance to degrade as more data is inserted into the database.

Data skew
The method should adjust for skewness in the data when performing the decomposition.

Geometrical properties
The size and the shape of the query region should have minimal impact on the response time when the size of the result set remains constant.

Number of ranges
The method should not generate more ranges than necessary to reduce the overall response time.

3.3 Best-first Decomposition

The proposed method is a quadtree-based, best-first decomposition of the query region. The query region is decomposed recursively with a hyper-quadtree to prune as much space as possible. The region is decomposed into intersecting and contained Z-ranges with rectangle-in-polygon tests. Points within contained ranges are added directly to the result set while points within contained ranges are refined. The refinement process validates each fetched point with a point-in-polygon test. To handle data skew, our method stores unprocessed intersecting hyper-quadrants in a priority queue instead of a normal queue (see the pseudocode of the breadth-first decomposition Algorithm 2 in the previous chapter). The priority value assigned to an intersecting hyper-quadrant is an estimate of how much data is stored within the range of the hyper-quadrant. The next hyper-quadrant to expand is determined to be the hyper-quadrant that contains the most data. The decomposition stops when either a maximum number of ranges are created, or the highest priority value in the priority queue is below a specific threshold.

Using a best-first decomposition allows us to create exact ranges for complexly shaped query regions and skewed data without overwhelming the system. By restricting the decomposition procedure to create precise ranges for the dense portions and coarser ranges for sparse portions of the query region, we can both reduce the total number of ranges significantly and achieve a higher range precision compared to a breadth-first decomposition when dealing with skewed data. This method also makes it possible to short-circuit the decomposition of low volume queries.

Estimating the priority value of the intersecting hyper-quadrants can be done in various ways. A possible approach is to use an overlying linear quadtree index like MD-HBase. Data within a hyper-quadrant can then be estimated accurately by counting the number of times its Z-value representation is a prefix match in the linear quadtree index. However, this approach comes with the cost of storing and maintaining an overlying index, which was seen to reduce insertion performance significantly in MD-HBase. Since we want to reduce the overhead of estimating the distribution, we instead use a sampled histogram over the stored Z-value keys and their prefixes. The reason for using a

histogram-based method is for its simplicity and the non-uniform nature of spatial and spatiotemporal data sets, which makes it more challenging to use precise parametric estimators, which place assumptions on the underlying distribution of the data.

The structure of the histogram will be similar to the structure of the linear quadtree with internal nodes in addition to the leaf nodes. Each node is associated with a counter that tells the number of points contained by the space covered by the node. The priority value of a hyper-quadrant can be estimated by looking up its Z-value representation directly with its associated counter in the histogram. Since the histogram needs only to provide an approximate representation of the data distribution to detect data skew and guide the decomposition, the histogram is built with sampling saving a significant amount of resources. Additionally, if the underlying distribution of the data changes slowly enough, then the construction and maintenance processes of the histogram can be decoupled entirely from the ingestion process, and instead be performed as a periodic background task. Therefore, the effect on the insertion performance is minimal.

Chapter 4

Implementation

This chapter describes the implementation details of our prototype for the proposed method described in the previous chapter and is developed in Java.

4.1 Database Layer

The database layer comprises the basic data structures to store and retrieve key-value pairs. The spatial and spatiotemporal access structure and query processing methods are implemented on top of the database layer.

4.1.1 Approach

The requirements for our database layer is simple, which is to provide a B+tree index structure supporting arbitrary key-value pairs and range scans. The data structure should support persistence of large data collections, and the database system should be embedded as it will always run on the same machine as the application. A possible approach is to implement a B+tree structure from scratch. However, supporting persistence and re-balancing millions of objects can prove non-trivial and time-consuming. The focus of this thesis is the query planning stage, and as there exist multiple fast, simple persistence database engines for Java with B+tree structures, the natural choice is to use an existing database solution.

4.1.2 MapDB

The prototype uses MapDB [10] as the underlying database engine, which is an open source embedded Java database engine that provides a series of efficient concurrent access structures backed by disk storage or off-heap memory. The original goal of MapDB was to provide a fast and simple alternative to existing SQL databases. According to its author, it is one of the fastest Java databases available today and is used by companies, such as Twitter, Linkedin, and HP Labs [16]. Our reason for choosing MapDB is

its simplicity, performance, and it offers all the features needed for our prototype. Additionally, MapDB is not limited to JVM memory, which makes it easier to handle large datasets.

BTreeMap

The data structure of interest provided by MapDB is the `BTreeMap` collection, which is a lock-free concurrent variant of the B+tree, called B-Linked-Tree, based on the work of Lehman and Yao [26]. The same data structure can be found in popular databases, such as Postgres [12]. The data structure offers high performance and good scalability for a large number of small keys. All the basic operations, such as insertion, removal, update, and access, can be executed concurrently by multiple threads. The `BTreeMap` class and its iterators implement all the methods of the Java `Iterator` and `Map` interfaces. The support for delete operations is limited in this access structure as a delete will not collapse nodes and cause fragmentation. Removing all entries in a full tree only releases about 60% of the used space. However, this will not present an issue for our use case.

Memory-mapped Files

The underlying database engine MapDB uses memory-mapped files to store and access data, which contains stored data in virtual memory and makes it appear as the entire database is loaded into memory. This permits applications to read and modify data directly through the primary memory. Memory-mapped files enable lazy loading by the operating system as accessed data is loaded into memory with demand paging. The operating system will also manage and keep recently accessed pages in memory. The result is increased I/O performance, especially in the case of larger files. However, 32-bit operating systems will be limited to 4GB of virtual memory by the addressing space. As our prototype will be used on a 64-bit operating system, which allows petabytes of virtual memory, this will not be an issue. Trashing may occur if the working set (i.e., the data set that is constantly requested) cannot fit into primary memory, which severely degrades performance since the operating system will continuously be swapping pages between memory and the disk.

4.2 Access Structures

To store and access spatial and spatiotemporal data efficiently with a B+tree index, we must first implement a preprocessor that linearizes our data with a space-filling curve.

4.2.1 Z-address Calculation

The data is linearized by use of Z-address keys, which are Z-values of maximum precision, i.e., cell level.

Definition 1. *(Z-address). Let the object $o \in$ n-dimensional space ω be represented with the coordinate attributes a_i and let the binary representation of each coordinate attribute a_i be denoted as $a_i = a_{i,s-1}a_{i,s-2}...a_{i,0}$ where $1 \leq i \leq n$. The Z-address of object o equals*

then the function value

$$Z(o) = \sum_{j=0}^{s-1} \sum_{i=1}^{n} a_{i,j} 2^{jn+i-1} \tag{4.1}$$

The Z-address, $Z(o)$, is stored in various data types or formats, such as a long primitive, as a `byte[]` or a `BigInteger` object. Objects such as `byte[]` and `BigInteger` can store Z-addresses with arbitrary precision while a long primitive restricts the key to 64-bit, which affects the accuracy of the space-filling curve. However, the long primitive uses significantly less memory compared to `BigIntger` or `byte[]` since it has no class or object overhead. The combination of less waste of cache space and having a modern CPU supporting a word size of 64-bit makes the execution with long primitives faster. Since encoding and decoding Z-addresses is such a common operation in our prototype, we rely upon the long primitive to store the Z-addresses.

In addition to the various formats that can be used to store the Z-address keys, there also exist various strategies to encode the Z-address. Just as choosing the right data format can heavily affect the performance of the system, choosing the right algorithm is often even more important, particularly when choosing between algorithms of different complexity classes.

For-loop–based method

The for-loop–based method is the most straightforward implementation and the most obvious way of encoding Z-addresses. The algorithm works by looping over and bitwise shifting the input coordinates to obtain the Z-address, which is similar to the encoding procedure described in Chapter 2. This method offers the benefit of being simple to implement and understand. However, the number of steps required to calculate the Z-address equals $N \times S$, where S is the number of bits used to represent the coordinate attributes and N is the number of dimensions.

Algorithm 3 For-loop–based method for encoding M-bits Z-addresses

```
1: /* o be the n-dimensional point described by its attributes a_i where 1 ≤ i ≤ n */
2: s ← sizeof(a_1) /* Number of bits used to represent the coordinate values */
3: z ← 0
4: for j ← 0, s do
5:     for i ← 1, n do
6:         z ← z ∨ (a_i ∧ 1 ≪ j) ≪ j + i − 1
7: return z
```

"Magic Bits" method

The encoding and decoding procedure used in the prototype is based on the "magic bits" method [11], which is significantly faster than the for-loop–based method while being harder to understand and implement. The idea is to use a combination of bit shifts and certain bit patterns or masks, called "magic bits", to split bits from the coordinate values into the Z-address. The magic bits are special constants or masks that are commonly used to enhance performance in various computer operations. In our case,

the masks remove bits that are shifted to incorrect positions. A thorough explanation of the "magic bits" method, which is depicted in Algorithm 4, is provided here due to its importance in the prototype.

Algorithm 4 "Magic Bits" based method for encoding M-bits Z-addresses

```
1: /* o be the n-dimensional point described by its attributes a_i where 1 ≤ i ≤ n */
2: m ← {m_1, m_2, ..., m_k} /* Loads set of predefined bit masks */
3: s ← {s_1, s_2, ..., s_k} /* Loads set of shift constants */
4: for each attribute a_i ∈ o do
5:     for j ← k, 1 do
6:         a_i ← (a_i ∨ a_i ≪ s_j) ∧ m_j
7: z ← 0
8: for i ← 1, n do
9:     z ← z ∨ a_i ≪ i − 1
10: return z
```

For each iteration, line 6 of Algorithm 4 shifts the bits of a coordinate attribute with a logical left shift operator and copies them into empty space with an OR operator. The masks m remove extraneous bits positioned in places to be used by the other coordinate attributes. The bits of each coordinate attribute are moved to their correct position by using successive power-of-two distance shifts. After the bits are spread correctly, the coordinate attributes are interleaved into the final Z-address.

This algorithm is best demonstrated through an example as shown in Figures 4.1 through 4.5 in which we encode three 10-bit coordinate attributes into a 32-bit 3D Z-address. Starting with the coordinate attribute $a_1 = [b_9, b_8, b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0]$, we spread its bits out to every third bit of a 32-bit address. The specific distances we spread the different bits in a_1 to are shown in Figure 4.1. The first pass of line 6 is a 16-distance shift. Looking at Figure 4.1, we observe that only bit b_9 and b_8 in a_1 need to move this far. Thus, bit b_9 and b_8 move 16 positions left while the mask ensures the other bits remain in their original position. Bit b_8 is now in its final position, so the remaining masks make sure that bit b_8 stays fixed during the remaining iterations. Bit b_9 must, however, travel two additional left positions, which is performed in the last iteration, which is a 2-distance shift. Now, looking at bit b_7 in Figure 4.1, it needs to move 14 positions left to reach its final position. Since we are only applying power-of-two distance shifts, we must move bit b_7 three times with an 8-distance shift (second iteration), a 4-distance shift (third iteration), and a 2-distance shift (last iteration). The same procedure is applied to the rest of the bits, as seen in the evolution from Figure 4.2 to Figure 4.5.

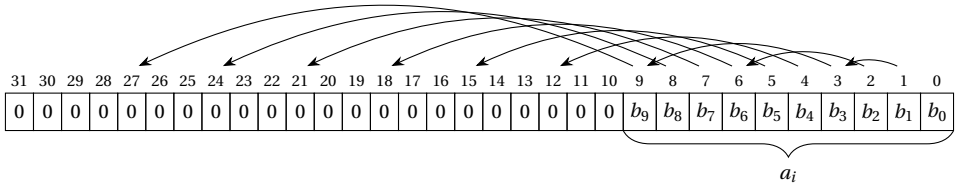


Figure 4.1: The specific distances we want to move the different bits in coordinate a_1

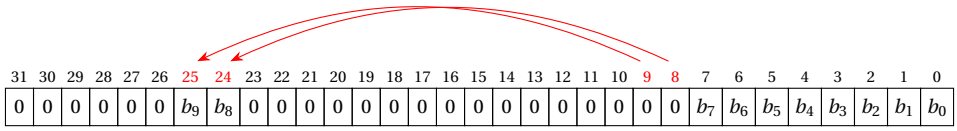


Figure 4.2: First pass is a $2^4 = 16$ distance shift. Only bit b_8 and b_9 in a_1 needs to move this far. The bit mask ensures that rest of the bits in a_1 stays unshifted.

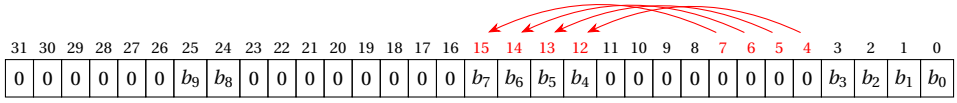


Figure 4.3: Second pass is a $2^3 = 8$ distance shift. Bit b_4, b_5, b_6 and b_7 in a_1 are shifted while rest of the bits stays in their position.

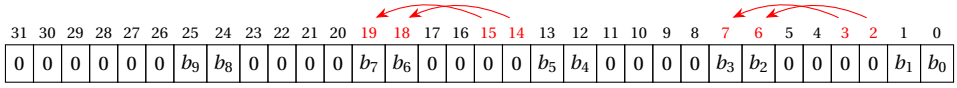


Figure 4.4: Third pass is a $2^2 = 4$ distance shift.

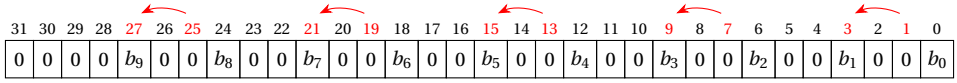


Figure 4.5: Fourth and last pass is a $2^1 = 2$ distance shift. All bits in a_1 are now spread out correctly.

Once all the coordinate values are spread out like this, they can easily be interleaved into the final Z-address. The Z-address is obtained in line 9 by shifting the coordinate values, so their bits fall into the empty space of each other. In other words, the bits in a_2 and a_3 are shifted one and two positions to the left, respectively, and combined with a_1.

The set of shift values, s, which are successive power-of-two values, and set of bit masks, m, which define the bits we want to keep after every shift, depend on the number of dimensions and bit precision. The bit masks and shift values used to encode 64-bit 2D, and 3D Z-addresses are given in Table 4.1.

Shift distance		Bit mask	
2D	3D	2D	3D
1	2	0x5555555555555555	0x1249249249249249
2	4	0x3333333333333333	0x10C30C30C30C30C3
4	8	0x0F0F0F0F0F0F0F0F	0x100F00F00F00F00F
8	16	0x00FF00FF00FF00FF	0x1F0000FF0000FF
16	32	0x0000FFFF0000FFFF	0x1F00000000FFFF

Table 4.1: List of successive bit shifts and bit masks that are used to encode 64-bit 2D and 3D Z-addresses.

Decoding a Z-address with "magic bits" is performed by executing the Algorithm 4 in reverse where the left shift operator is changed with a right shift operator, and the order of the shifts and the masks are changed. Computing the Z-address using the "magic bits" method takes at most $N \times \lceil log_2 S \rceil + N$ steps, which makes the "magic bits" method significantly faster than the for-loop–based method. Given the benchmark result presented by [11], it is an order of magnitude faster when encoding 64-bit random 3D Z-addresses compared to the for-loop–based method. There exists a slightly more efficient divide-and-conquer method that uses pre-computed tables that contain the correct splits for certain bit inputs. The performance improvement is, however, too low to justify the additional implementation effort.

4.2.2 Encoding of Space and Time

Before we calculate Z-addresses, we need to consider the encoding of the spatial input coordinates and the timestamp. Choosing the appropriate encoding is essential to utilize the space-filling curve efficiently. First, the dimensions of the space-filling curve should be of similar significance during decomposition. In other words, the queried dimensions should be of similar length. Second, the dimensions should be scaled with respect to the available bits. This prevents us from wasting large portions of the space-filling curve and making the decomposition more inefficient.

Space

The spatial coordinates of the data points are presented as a latitude and longitude pair with the ranges [-90, 90] and [-180, 180], respectively. The spatial coordinates are read from the data point and converted to the appropriate format to be used for calculating the Z-address. The conversion is done by applying the following linear transformation:

$$x_s = \frac{(x+90)}{90} \times 2^{bits\ per\ dimension} \tag{4.2}$$

$$y_s = \frac{(y+180)}{180} \times 2^{bits\ per\ dimension} \tag{4.3}$$

, where x is the latitude and y is the longitude. The spatial index uses 31 bits per dimension while the spatiotemporal index uses 21 bits per dimension. The reason for using 31 instead of 32 bits per dimension is because Java does not support unsigned number types, so the last bit is used for the sign. A precision of 21 bits gives a latitudinal granularity of 0.000087 degrees and a longitudinal granularity of 0.00017 degrees, which is roughly equivalent to a precision of 100 meters. The precision of the spatial index is roughly 10 centimetres.

Time

The timestamps of the data points are represented in the format $yyyy/MM/dd-hh/mm/ss$. Choosing the appropriate format for the timestamp is not as trivial as with the spatial coordinates as the main issue is that we are working with an unbounded dimension. The temporal dimension needs to be restricted to an interval of fixed length as the length of the space-filling curve is fixed. Thus, all the encoded timestamps need to fall within some specified interval $[t_{start}, t_{end}]$, this size of which defines the range of allowable timestamps. A possible approach is to pick a very wide range that allows storing data in all the foreseeable future (e.g., 1,000 years). This approach, however, will waste large portions of the space-filling curve and only allow for a very coarse time resolution, which makes queries over small time windows inefficient. A narrow range allows for a higher time resolution and utilizes the curve better, but may cause overflow problems for future data.

Another approach that avoids the issue of having an unbounded dimension uses a concept called periodicity, which is employed by GeoMesa. The idea is to bin the space-filling curve into a period block, which can be a day, a week or a year, for example. Time in the space-filling curve is then the offset into the period block. Our B+tree index stores multiple space-filling curves one for each period block indexed. This approach comes with more overhead but allows a very fine temporal resolution without having to worry about possible overflow. The period block should ideally not be larger than the time size of the query. If the data is partitioned into day-sized blocks and we query a week-long interval, then we would execute a separate query for each day of the queried week.

The following two approaches are employed and tested in this thesis.

1. **Integrated approach**. This approach has all data inserted into a single space-filling curve. For performance reasons, we set the temporal range of the curve equal to that of the test data. This solution is not ideal for applications that include unbounded dynamic data sets. The timestamps are converted by applying the following linear transformation:

   $$z_s = \frac{z}{t_{max}} \times 2^{bits\ per\ dimension} \tag{4.4}$$

   , where z is the number of minutes between the inserted data point and the minimum timestamp in the test data set, and t_{max} is the maximum timestamp in the test data set.

2. **Semi-integrated approach**. This approach relies on periodicity where the index key includes a period number, which is also called the epoch, with the Z-address. It is chosen to partition the data into year-long space-filling curves as it demonstrates the best performance in our initial testing. The epoch is defined to be the number of years passed since the Java Epoch (*1970-01-01T00:00:00*), then the timestamps are converted by applying a similar linear transformation as the integrated approach. The difference is that t_{max} is now the maximum number of minutes in a year and z is the number of minutes between the inserted data point and the date *yyyy-01-01T00:00:00*, where *yyyy* is the year of the inserted data point.

4.2.3 Indexes

Three difference indexes are used in this thesis. The first is the spatial index used for spatial queries, which is also called the xy index. The next two indexes are the spatiotemporal indexes, which both support the same spatiotemporal queries. The difference is how time is incorporated into the space-filling curve. The integrated spatiotemporal index, which is called the xyz index, integrates time fully in the space-filling curve. The semi-integrated spatiotemporal index, which is called the $t-xyz$ index, creates a separate space-filling curve for each year indexed where time in the curve is the offset into the year. Within the B+tree structure, keys are first sorted by the epoch and then by the Z-address. The different key formats are depicted in Figure 4.6.

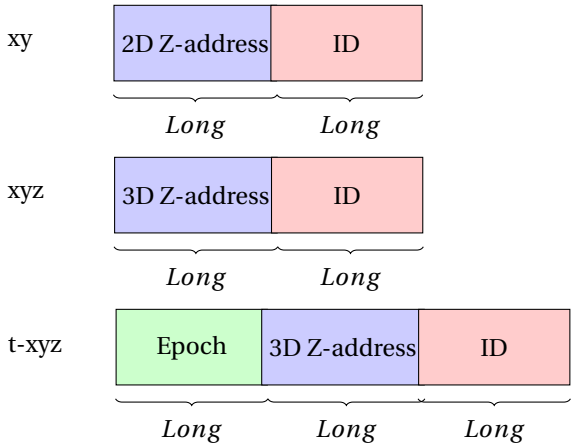


Figure 4.6: A more detailed representation of the index keys used in the prototype.

The keys also contain a unique ID, which is a sequence number that prevents the existence of duplicate keys. Duplicates are not allowed by the underlying database engine, MapDB, and there exists a chance that data points in proximity in space and time are mapped to the same Z-address.

4.2.4 Insertion Procedure

The insertion procedure of a data point is schematically depicted in Figure 4.7. The preprocessing stages are summarized as follows:

1. *Extract.* The space (latitude and longitude) and time (timestamp) attributes are extracted from the data point.
2. *Encode space and time.* The encoding procedure depends on the index. The latitude and longitude are scaled to use 31 bits for the spatial index and 21 bits for the spatiotemporal index. The time is converted into the offset of the temporal range of the space-filling curve before scaled to 21 bits.
3. *Calculate Z-address.* The Z-address is calculated with the encoded space and time attributes.
4. *Create key.* The key is generated by concatenating the Z-address and the sequence number, which is incremented during each insertion. For the semi-integrated index, we concatenate the epoch with the Z-address and the sequence number.

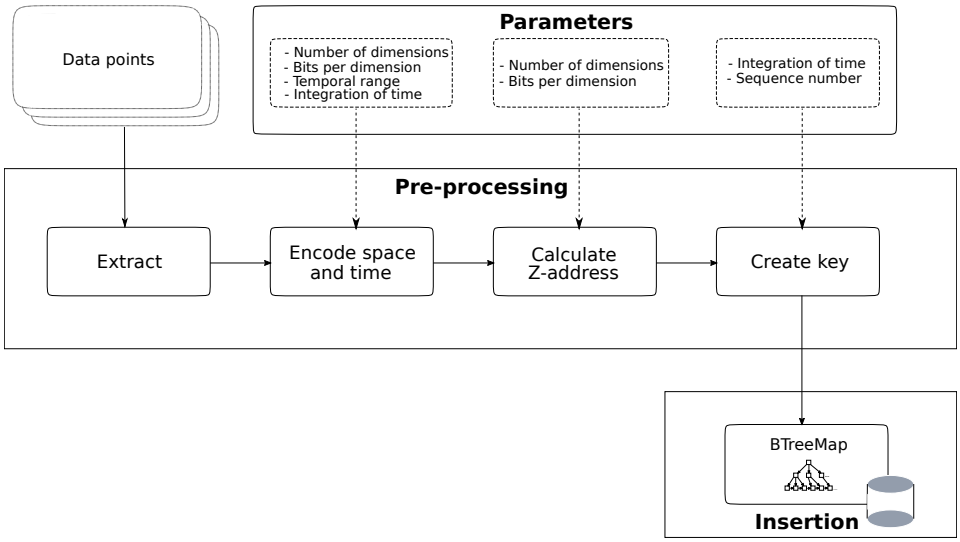


Figure 4.7: Overview of the insertion procedure.

4.3 Query Handling

As the access method provided by the underlying database layer is not sufficient to support efficient retrieval of spatial and spatiotemporal data, we must implement an additional query processor on top of the database layer. This includes our proposed method of decomposing region queries into sets of one-dimensional range scans. Before describing the implementation details, the supported region queries are specified in more detail in the following.

Spatial Region Queries. A spatial region query returns all the points contained within a simple polygon. The simple polygon must be represented as a set of coordinate vertices. The polygon cannot contain self-intersecting line segments or holes.

Spatiotemporal Region Queries. A spatiotemporal region query returns the points contained within a simple polygon and a bounded time range.

The process of executing a spatial or spatiotemporal region query is divided into three steps. First, we find Z-ranges that intersect or overlap with the query region in space and time, which is also known as the decomposition stage. The second stage is the filter where the Z-ranges from the previous stage are executed as range scans against the linearized B+tree index. Points retrieved from ranges that are fully covered by the query region are added to the result set. Points retrieved from ranges that intersect with the query region are sent to the next stage. The third refinement stage validates all points retrieved from intersecting ranges. Points that pass the validation test are added to the result set, and false positives are discarded. The complete process is schematically depicted in Figure 4.8.

The next sections describe the three stages in more detail.

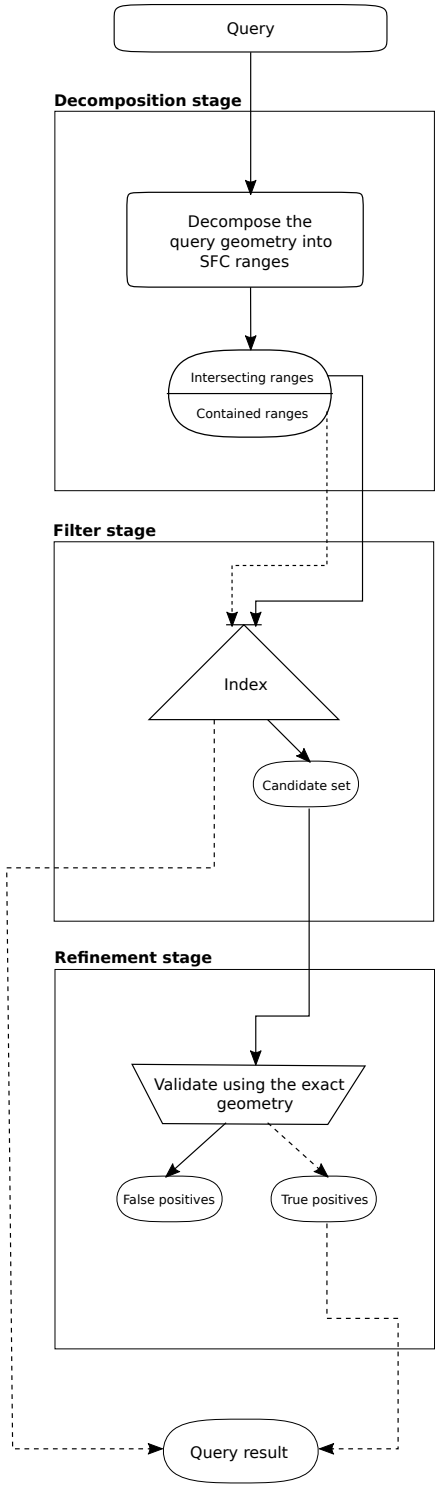


Figure 4.8: Overview of the query processing procedure.

4.3.1 Decomposition Stage

The decomposition stage is the first stage of executing a region query. As mentioned earlier, processing region queries efficiently require first decomposing the query region in N-dimensional space into a set of one-dimensional ranges on the space-filling curve. The goal of the decomposition process is to minimize the response time by efficiently reducing the amount of data that needs to be checked and removed in the refinement stage.

The query processor takes an array of coordinate vertices and a time interval as input parameters. The first parameter is required for both spatial and spatiotemporal. The second parameter is optional and used for spatiotemporal queries. Before running the decomposition algorithm, the following preparation steps must be performed.

1. The coordinate vertices are encoded with the same encoding scheme applied during the insertion process to calculate Z-address keys.
2. (Optional) The time interval is encoded and converted to a minimum and maximum offset value.
3. (Optional) If a query is executed against the semi-integrated spatiotemporal index and the time range of the query spans multiple epochs, then we execute a separate decomposition for each epoch.

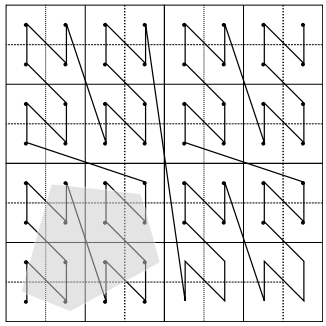
The set of encoded coordinate vertices are used to instantiate a geometry object p_{query} from the package `org.locationtech.jts.geom`. The prototype relies heavily on the JTS Topology Suite library [9] for the spatial data operations. The instantiated geometry object is used to test if the hyper-quadrants spatially intersect or are covered by the query region. The fact that the same geometry object is repeatedly used allows us to optimize the performance significantly. When JTS performs a predicate test, it creates an index structure, called an edge graph, for the geometry. To avoid building the same edge graph repeatedly, it can be computed once and cached for reuse. Caching the edge graph is enabled by explicitly creating a `PreparedGeometry` on the geometry object, which additionally enables detection of certain situations where the predicate test can be short-circuited.

The pseudocode of the implemented best-first decomposition algorithm is presented below. The algorithm is based on the proposed method in Chapter 3 and is explained in depth below.

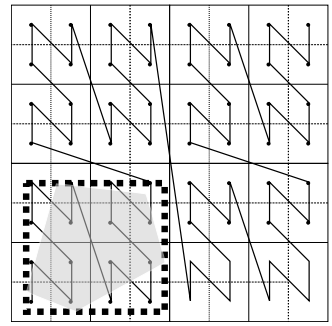
Algorithm 5 Quadtree-based best-first decomposition of a query region.

1: /* Q_N is the scaled and encoded query region, where N = 2 means the query is spatial and N = 3 is spatiotemporal. */
2: $\mathbb{S} \leftarrow \emptyset$ /* Initialize an empty set to store result ranges.*/
3: $R \leftarrow \emptyset$ /* Initialize an empty PriorityQueue to store h-quadrants to process. */
4: $p_{query} \leftarrow jts\_spatial\_polygon(Q_N)$
5: $p_{MBR} \leftarrow jts\_spatial\_envelope(p_{query})$
6: $\{x_{lower,query}, y_{lower,query}, x_{upper,query}, y_{upper,query}\} \leftarrow coordinates(p_{MBR})$
7: **if** $N = 3$ **then**
8: $\quad \{t_{lower,query}, t_{upper,query}\} \leftarrow temporal\_offset\_bounds(Q_N)$
9: $z_{lower} \leftarrow z\_address\_encode(x_{lower,query}, y_{lower,query}, t_{lower,query})$
10: $z_{upper} \leftarrow z\_address\_encode(x_{upper,query}, y_{upper,query}, t_{upper,query})$
11: $z_{prefix,root} \leftarrow common\_prefix(z_{lower}, z_{upper})$ /* root hyper-quadrant */
12: $ENQUEUE(R, z_{prefix,root}, max\_priority)$
13: **while** $priority(PEEK(R)) > threshold$ and $|\mathbb{S}| + |R| < maximum\_ranges$ **do**
14: $\quad z_{prefix} \leftarrow DEQUEUE(R)$
15: $\quad$ **for** each sub-h-quadrant $z_{prefix,sub}$ in z_{prefix} **do**
16: $\quad\quad \{z_{lower}, z_{upper}\} \leftarrow z_{prefix,sub}$
17: $\quad\quad \{x_{lower}, y_{lower}, t_{lower}\} \leftarrow z\_address\_decode(z_{lower})$/* $t = \emptyset$ if $N = 2$*/
18: $\quad\quad \{x_{upper}, y_{upper}, t_{upper}\} \leftarrow z\_address\_decode(z_{upper})$
19: $\quad\quad p_{quadrant} \leftarrow jts\_spatial\_polygon(x_{lower}, y_{lower}, x_{upper}, y_{upper})$
20: $\quad\quad$ **if** $CONTAINS(t_{lower}, t_{upper}, t_{upper,query}, t_{lower,query}$
21: $\quad\quad\quad\quad , p_{quadrant}, p_{query})$ **then**
22: $\quad\quad\quad \mathbb{S} \cup \{z_{lower}, z_{upper}\}_{Contained}$
23: $\quad\quad$ **else if** $INTERSECTS(t_{lower}, t_{upper}, t_{upper,query}, t_{lower,query}$
24: $\quad\quad\quad\quad , p_{quadrant}, p_{query})$ **then**
25: $\quad\quad\quad ENQUEUE(R, z_{prefix,sub}, priority(z_{prefix,sub}))$
26: **while** $R \neq \emptyset$ **do**
27: $\quad z_{prefix} \leftarrow DEQUEUE(R)$
28: $\quad \{z_{lower}, z_{upper}\} \leftarrow z_{prefix}$
29: $\quad \mathbb{S} \cup \{z_{lower}, z_{upper}\}_{Intersecting}$
30: **return** $\mathbb{S}$

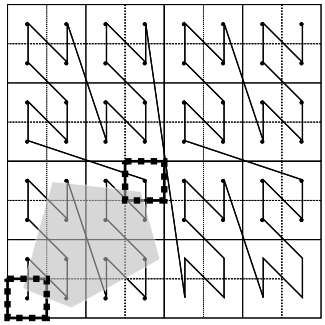
To start the quadtree-based decomposition of the query region, we first calculate the initial root node of the hyper-quadtree. The root node is set to be the minimum covering hyper-quadrant of the query region and provides an initial restriction of the space to be decomposed. As described in Chapter 2, the relationship between the Z-order curve and the hyper-quadtree lets us represent a hyper-quadrant as a Z-address prefix (z_{prefix}). The minimum covering hyper-quadrant of the query region is the longest common Z-address prefix of the Z-addresses that represents the lower-left and upper-right vertices of the minimum bounding hyper-rectangle of the query geometry in N-dimensional space. The process of calculating the root node for a spatial region query is depicted in Figure 4.9.



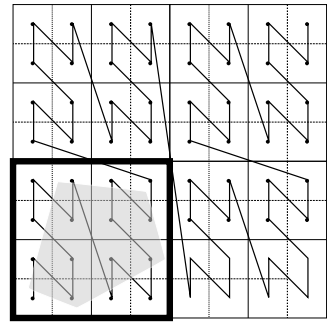
(a) The query region.



(b) MBR of the query region.



(c) The Z-addresses (see figure 4.9) of the lower-left (000000) and upper-right (001111) vertices of the MBR.



(d) The quadrant represented by the longest common Z-address prefix (00) of the lower-left and upper-right vertices of the MBR.

Figure 4.9: An example of how the root node is calculated for a spatial region query.

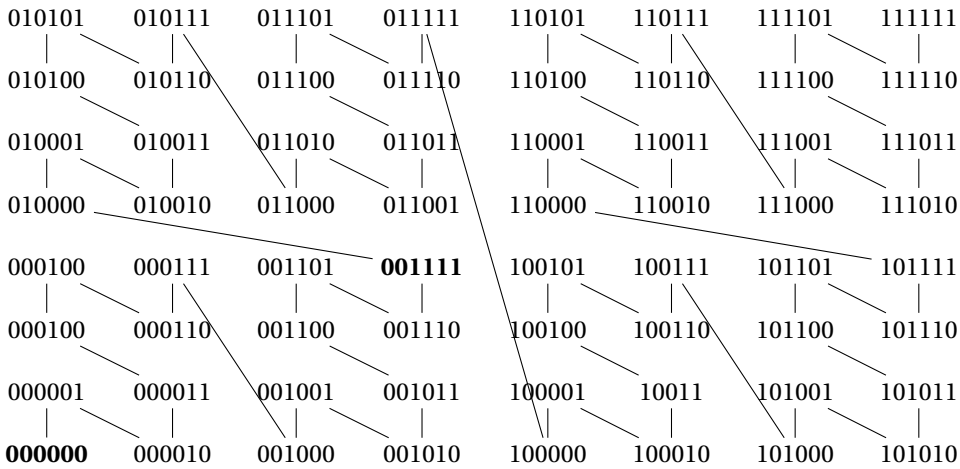


Figure 4.10: The Z-addresses for a two-dimensional Z-order curve with 3 bits per dimension.

The algorithm picks the next hyper-quadrant to expand from a priority queue. The first hyper-quadrant to expand is the root node, which is expanded into 2^N smaller hyper-quadrants by appending all possible N-bit combinations to the Z-address prefix (where N is the number of dimensions). Each sub-hyper-quadrant is evaluated by testing if it is contained or intersects with the query region.

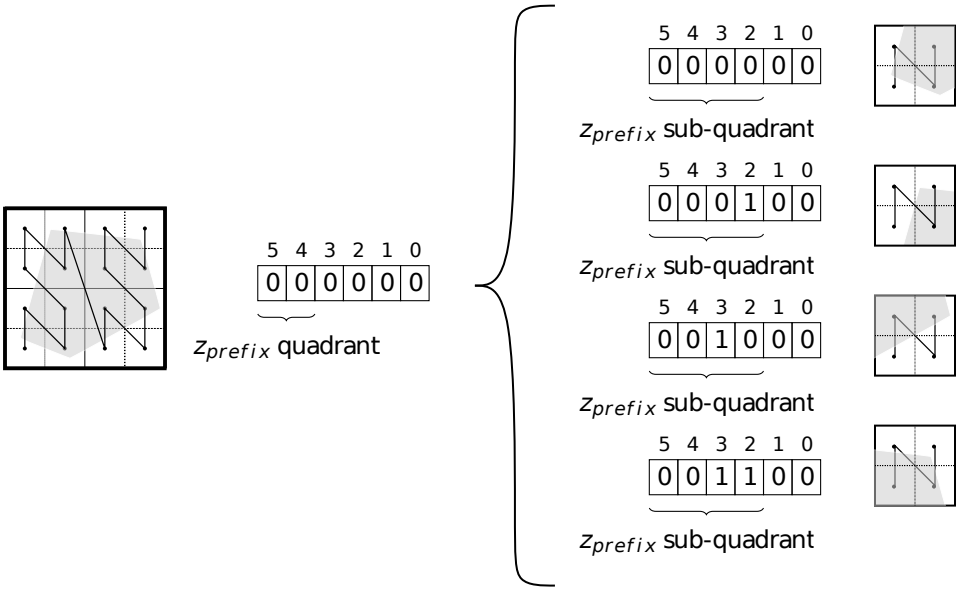


Figure 4.11: Expanding the example root node into $2^2 = 4$ sub-quadrants.

Contained Hyper-quadrants

To determine if a hyper-quadrant intersects or is contained by the query region, we first calculate its lower bound z_{lower} and upper bound z_{upper}. The lower bound z_{lower} is the z_{prefix} followed by trailing bits of zeros, and the upper bound z_{upper} is the z_{prefix} followed by trailing bits of ones.

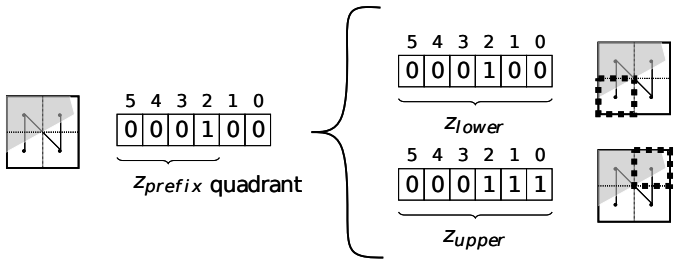


Figure 4.12: Calculating the Z-bounds z_{lower} and z_{upper} by setting the bits after the prefix to zero and one, respectively.

The Z-addresses z_{lower} and z_{upper} are decoded to their coordinate values in N-dimensional space, which represent the lower-left and upper-right vertices, respectively, of the hyper-quadrant in N-dimensional space. A geometry object $p_{quadrant}$ of the hyper-quadrant in the spatial plane is instantiated with the spatial coordinates x_{lower}, y_{lower}, x_{upper} and y_{upper}.

The contained test includes temporal and spatial tests. The temporal test is performed first for spatiotemporal queries, and checks if the temporal coordinates t_{lower},t_{upper} of the hyper-quadrant are contained by the time interval of the query: $[t_{lower}, t_{upper}] \subseteq [t_{lower,query}, t_{upper,query}]$. If it passes the temporal test or if the query is spatial, then a more computationally expensive spatial test is performed. The `contains(geometry)` method from the JTS `geometry` class is used to determine if the spatial geometry object $p_{quadrant}$ is contained by the geometry object p_{query}. JTS will automatically detect if the $p_{quadrant}$ geometry is a rectangle and perform a rectangle-in-polygon test instead of a more expensive polygon-in-polygon test. If it passes the spatial intersection test, then the range of the hyper-quadrant, $\{z_{lower}, z_{uppper}\}$, is marked as contained and added to the result set of ranges. Hyper-quadrants that are not contained by the query are passed on to the intersection test.

Intersecting Hyper-quadrants

The intersection test checks first if either t_{lower} or t_{upper} is within the interval $[t_{lower,query}, t_{upper,query}]$. If the hyper-quadrant temporally overlaps with the query or the query is spatial only, then the `intersects(geometry)` method from the JTS `geometry` class is used to check if the hyper-quadrant geometry spatially intersects with the query geometry. Hyper-quadrants determined to intersect with the query region are stored in the priority queue for further expansion. Intersecting hyper-quadrants are assigned a priority value according to the estimated data distribution when inserted into the priority queue. Hyper-quadrants estimated to contain a significant amount of data are assigned a higher priority value than hyper-quadrants containing less estimated data. The hyper-quadrant with the highest priority value is selected in the next iteration of the decomposition algorithm.

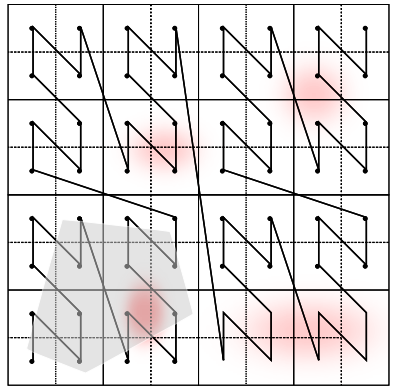


Figure 4.13: The red color shows the data distribution in the example space.

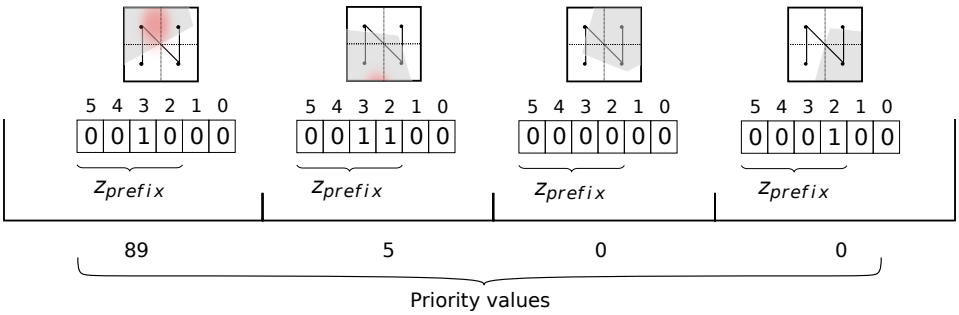


Figure 4.14: The priority queue after the first iteration. The priority value reflects the amount of data contained in the quadrant.

Hyper-quadrant Priority

The priority value of intersecting hyper-quadrants is determined by a histogram, which is implemented by using a hash table that provides linear complexity in terms of constructing cost and memory. The lookup time is, however, constant. Hash tables make it possible to construct and update the histogram incrementally. Updating the histogram with new data will, however, increase the size, and, thus, the memory cost of the hash table. This problem can be mitigated by rebuilding the hash table with sampling. Our implementation relies on the `Long2LongOpenHashMap` class provided by the fastutil library, which is built specifically for speed and a low memory footprint [3]. A separate histogram is built for each index, and each histogram is constructed using a sample of index keys. The entry of the hash table is a Z-address prefix (hash key) representing a hyper-quadrant and a frequency counter (hash value). The histogram is constructed incrementally with one sampled index key available at the time. The Z-address of the index key is partitioned into $\lfloor \frac{M}{N} \rfloor$ prefixes of increasing length, where M is the Z-address bit-length, and N is the number of dimensions. The partitioning process is performed by increasing the prefix of the Z-address with N bits at the time until the root (base) is empty. The histogram is then updated by incrementing the frequency counter of each prefix in the hash table by one.

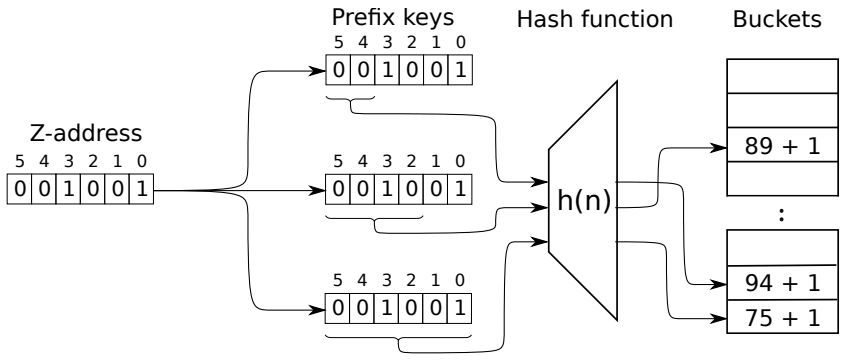


Figure 4.15: Updating the histogram with an index key.

The histogram can be thought of as a linear quadtree with interior nodes where each node is associated with a counter that tells how many points are contained within a hyper-quadrant. The process of updating the histogram with an index key is then equal to traversing a hyper-quadtree top-down to the leaf node that contains the index key while incrementing the counter to each of the visited nodes (only one node per level in the tree is updated).

If no entry is found in the hash table for an intersecting hyper-quadrant during decomposition, it will be assigned a priority value of zero. The decomposition stops when the priority value of the next hyper-quadrant in the priority queue is below a specific threshold or a maximum number of ranges is created [1]. The remaining hyper-quadrants in the priority queue will be added to the result set as intersecting ranges. The priority threshold value must be tuned to balance the overhead from creating and executing additional range scans against the overhead from additional refinement.

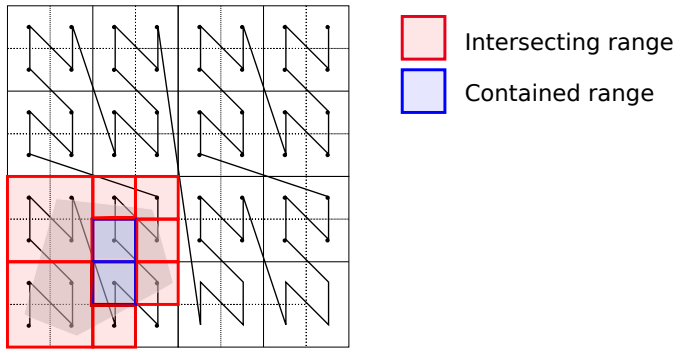


Figure 4.16: The complete decomposition of the example query. Notice that the hyper-quadrants $z_{prefix} = 0000$ and $z_{prefix} = 0001$ are not expanded since their priority value equals zero.

Merging Adjacent Ranges

A final post-processing step of merging adjacent ranges is applied to reduce the number of ranges in the result set $\mathbb{S}$. The idea is to join multiple consecutive ranges into larger continuous ranges resulting in fewer range scans during the filter stage. The merging procedure is the following with the ranges in the result set $\mathbb{S}$ first sorted in increasing order where

$$\{z_{i,lower}, z_{i,uppper}\} < \{z_{j,lower}, z_{j,uppper}\} \quad \text{when} \quad \begin{array}{l} z_{i,lower} < z_{j,lower} \vee \\ z_{i,lower} = z_{j,lower} \wedge z_{i,upper} < z_{j,upper} \end{array}$$

$$\{z_{i,lower}, z_{i,uppper}\} > \{z_{j,lower}, z_{j,uppper}\} \quad \text{when} \quad \begin{array}{l} z_{i,lower} > z_{j,lower} \vee \\ z_{i,lower} = z_{j,lower} \wedge z_{i,upper} > z_{j,upper} \end{array}$$

$$\{z_{i,lower}, z_{i,uppper}\} = \{z_{j,lower}, z_{j,uppper}\} \quad \text{when} \quad \begin{array}{l} z_{i,lower} = z_{j,lower} \wedge \\ z_{i,upper} = z_{j,upper} \end{array}$$

[1] The best-first decomposition procedure will mainly be stopped by reaching the priority threshold.

The merging procedure is then performed by iterating over the sorted set and merging the current range Z_{curr} with the next range Z_{next} in the set if the next lower bound and current upper bound are consecutive numbers:

$$Z_{curr} = \begin{cases} \{z_{curr,lower}, z_{next,uppper}\} & \text{if } (z_{curr,upper} + 1) = z_{next,lower} \\ Z_{next} & \text{otherwise} \end{cases} \tag{4.5}$$

The merging process starts by setting $Z_{curr} = Z_0$ and performs a total of $|\mathbb{S}|$ iterations. The time complexity of the algorithm is, however, $O(|\mathbb{S}| \log |\mathbb{S}|)$ due to the sorting. Intersecting ranges are only merged with other intersecting ranges while contained ranges are only merged with other contained ranges. If an intersecting range were merged with a contained range, then the result would be a larger intersecting range that requires additional refinement for false positives. The final merging step is meant to reduce the overhead of searches in the B+tree index and will not affect the accuracy of the decomposition and the number of false positives retrieved.

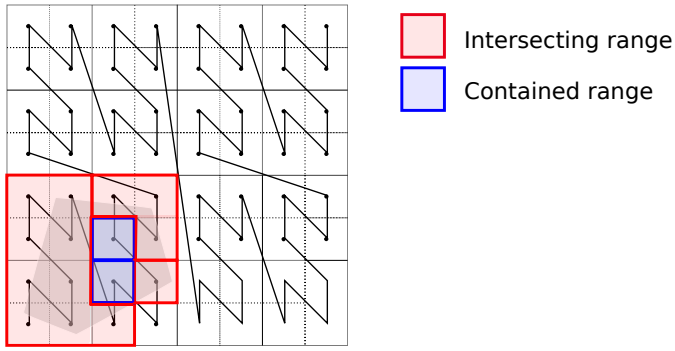


Figure 4.17: The number of ranges in the decomposition in figure 4.16 can be reduced from nine ranges to five ranges by merging adjacent ranges. The quality of the decomposition remains the same.

4.3.2 Filter Stage

The filter stage executes a series of range queries against the B+tree index by using the generated ranges from the decomposition step. A range query is executed by creating an interval submap with the `subMap(lower, upper)` method provided by the Java `NavigableMap` interface implemented by the `BTreeMap` class. The submaps (ranges) are partitioned into two separate sets of contained submaps and intersecting submaps. The objects fetched from the contained ranges are stored directly in the result set while objects fetched from the intersecting ranges are stored in a candidate set. The objects in the candidate set may contain false hits and need to be validated. The objects are loaded lazily as the data itself are accessed, and the objects in the candidate set are first fetched during the refinement stage.

4.3.3 Refinement Stage

The refinement stage inspects the exact representation of each object in the candidate set from the filter stage. The coordinates and the timestamp information are extracted

from the retrieved object and tested against the exact spatial query geometry and potential time interval. The spatial test is a point-in-polygon (PIP) test provided by JTS. Only objects that pass the spatial and the potential temporal test are stored in the result set. The refinement stage is usually the most computationally expensive step of the query execution because of the potential number of PIP operations required. The PIP implementation in JTS is based on the well-known ray-casting algorithm that draws a line from the point and counts how many times the line intersects with the polygon. The point is determined to be inside the polygon if the drawn line intersects with the polygon an odd number of times. Otherwise, if it intersects an even number of times, then the point is outside. The original algorithm usually runs with linear complexity. However, logarithmic performance is possible in cases of the repeated testing with the same polygon, which applies in our application. This performance increase is achieved by using the `PreparedGeometry` class to build and cache a pre-computed edge graph for the query polygon.

4.4 Parallelization

The query handling algorithm described in this chapter has a high potential for parallelization, which can be exploited on modern multi-core processing systems to increase query performance. Parallel processing is employed in the decomposition and refinement stages.

4.4.1 Multi-threaded Decomposition

The fact that the space covered by intersecting hyper-quadrants in the priority queue are not overlapping makes it trivial to parallelize the decomposition to a nearly arbitrary level. A possible approach for parallelization is partitioning the root node of the hyper-quadtree into multiple intersecting hyper-quadrants and assigning each thread one of the intersecting hyper-quadrant as a root node, as seen in Figure 4.18. Each thread decomposes a separate portion (a sub-hyper-quadtree) of the query region. The range sets created from each thread are merged at the end, which includes merging adjacent ranges. No synchronization is required between the threads during decomposition.

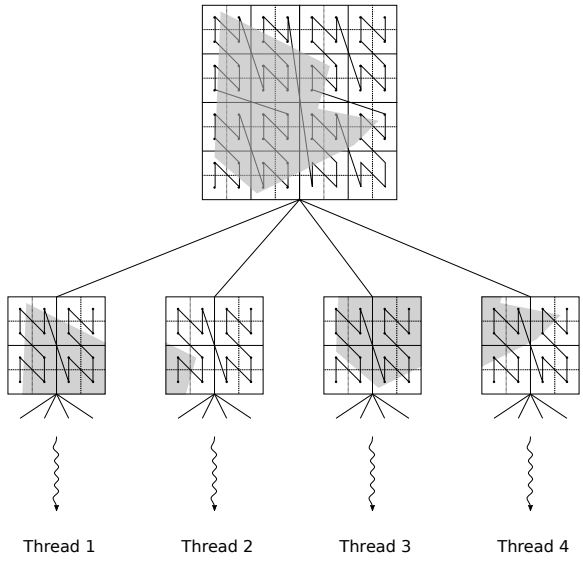


Figure 4.18: Simple multi-threaded decomposition.

This approach does, however, have shortcomings. Specifically, there is a significant chance of load imbalance between the threads. This can be seen in threads that are assigned hyper-quadrants that intersect only a small portion of the query region and covers sparse regions. These threads are likely to finish early compared to threads that are decomposing dense regions that require significantly more ranges to reduce the number of false positives. To alleviate this issue, we perform an initial shallow decomposition of the query region. The intersecting hyper-quadrants in the priority queue are then partitioned into sets of approximately equal priority value sum. Each thread is assigned a set of intersecting hyper-quadrants, which are decomposed in parallel. The priority value sum of a set proves to be a good indicator of how many ranges are likely to be created for the specific set. The load balance is, thus, improved at the cost of a minor reduction in concurrency. The initial decomposition is very shallow, so the potential loss in concurrency is expected to be insignificant in most use cases. The problem of partitioning the priority queue into subsets with the equal priority value sum can be reduced to the well-known NP-complete partitioning problem. This problem is partially solved by assigning every *nth* hyper-quadrant in the priority queue, which is sorted by the priority value, to the *nth* thread. This simple approach demonstrates sufficient load balancing in our initial tests.

4.4.2 Multi-threaded Refinement

Parallelization is implemented in the refinement stage by queuing up each of the intersecting ranges as a separate refinement task. A Java `ExecutorService` is instantiated with a fixed number of threads to reduce overhead due to thread creation. The number of instantiated threads should equal the number of processor cores available so each core can be fully utilized.

The refinement tasks are executed concurrently by calling `invokeAll()` on the list of tasks, and the results from each task are added to the final result set once all tasks are complete. The initial results show a high utilization of the available cores with this approach. Good load balance is achieved as the job queue consists of many short tasks, although the tasks are large enough to be worth the overhead of executing on a thread pool. Attempts were made to reduce overhead further by batching the tasks into larger tasks, but no performance improvements were identified.

Chapter 5

Experiments, Results, and Discussions

In this chapter, we design and execute a complete benchmark for our prototype. The purpose of the benchmark is to measure the performance in terms of scalability, response time, resource use and false positives.

5.1 Baseline

To analyze the performance of our prototype, it is necessary to establish a baseline system. For this purpose, we implemented the quadtree-based, breadth-first decomposition strategy introduced in Chapter 2, which is chosen as it is one of the most popular approaches of decomposing region queries into SFC ranges. Our implementation of the breadth-first decomposition method is based on the one provided by the LocationTech SFCurve library [5], on which GeoMesa relies. However, instead of partitioning the MBR of the query region into intersecting ranges with rectangle-in-rectangle tests, like GeoMesa, we partition the query region into intersecting and contained ranges with rectangle-in-polygon tests.

The breath-first decomposition method is static compared to the best-first decomposition method as it creates approximately the same number of ranges for each query. The exception is for particularly small polygons or axis-aligned boxes where the decomposition procedure hits the recursive limit imposed by the precision of the Z-address keys. The maximum number of ranges that will be created for a query is given by a parameter value.

5.2 Dataset

For all experiments performed in this chapter, the GeoLife dataset from Microsoft is used [36][35][34], which consists of real-life GPS trajectory data recorded from 178 different users with GPS-enabled devices. The data spans a four-year period, and each recording is stored as a timestamped point with latitude and longitude coordinates. The sampling rate is approximately $[1,5]$ seconds or $[5,10]$ meters, and the set contains a total of $20,283,945$ points. The data set is primarily recorded from users in China, and most of the data is located within Beijing. A heat map that shows the distribution of the recorded data is shown in Figure 5.1.

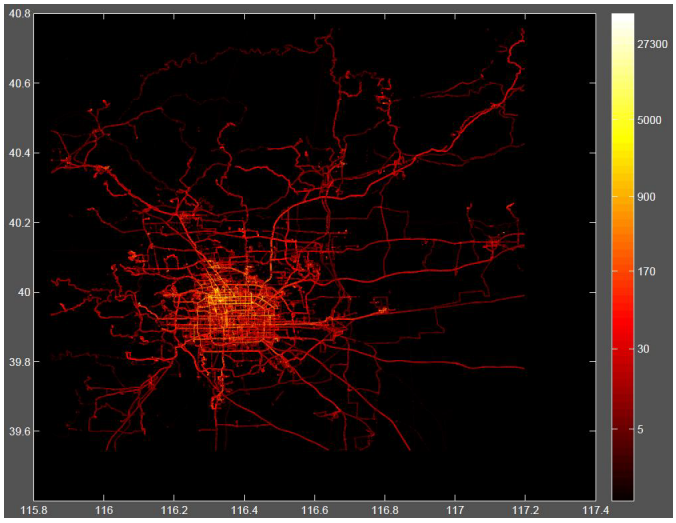


Figure 5.1: Distribution of data in Beijinig [34]

This dataset is chosen for our experiments as it represents real-world data over wide spatial and temporal ranges. It is likely to provide an accurate view of how our proposed method will perform in real use cases compared to using a computer-generated data set.

5.3 Queries

To test the performance of our prototype, a set of spatial and spatiotemporal queries is developed.

5.3.1 Spatial Region Queries

A total of 120 random generated polygons of different complexity and sizes within Beijing were used for the spatial query set. To determine if there is a significant difference between the baseline and the proposed method in our prototype, we ensured each polygon contains at least 10,000 data points. The spatial queries are meant to give us an insight into how the prototype behaves with different geometries and sizes.

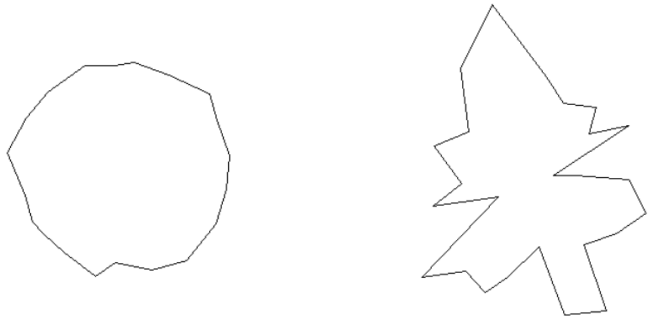


Figure 5.2: Two example polygons.

5.3.2 Spatiotemporal Region Queries

The spatiotemporal query set consists of 30 different polygons where each polygon has a total of 15 different temporal predicates giving a total of 450 spatiotemporal queries. Three different time sizes — day, week, and month — were used for the temporal predicates. The time predicates for each spatial predicate were selected mostly randomly across the time range of the GeoLife test set, and it was ensured that none of the spatiotemporal queries was empty.

The queries are executed sequentially in our experiments. To keep the results consistent and to normalize cache side-effects, we presented each query five times in a random order and averaged the results.

5.4 System Configuration

The specifications of the hardware used for all the experiments are listed in Table 5.1. In addition to the implementation details described in the previous chapter, there are configuration parameters for the underlying database layer. Most importantly, nodes in the B+tree are serialized, and each node contains at most 32 pointers. This is the default node size in MapDB and proved to provide acceptable performance in our initials tests. As changing the node size will predominantly affect random lookups and not sequential range scans, it was decided that it not be worth tuning the node size. Write-A-Head logging is also disabled as atomicity and durability are not of any concern in our experiments.

Operating System	Windows 10 64bit
Processor	Intel Core i5-3750k CPU @ 4.2 GHz
Logical Cores	4
Primary Storage	16GB
Secondary Storage	500GB (SSD)

Table 5.1: System configuration.

The amount of free physical memory in our system is greater than the total memory size of the indexes and data sets used in our experiments. This means that page reuse will be especially effective when using memory-mapped files as the entire dataset can be copied to memory as it is touched, and the operating system will not be forced to start aggressive unmapping of files because of page faulting. The consequence is that query processing is eventually going to be CPU-bound in our experiments.

5.5 Metrics

To evaluate the performance of our experiments, we defined a set of quantitative measures for the different performance aspects of the system.

Number of Ranges

The number of ranges is the number of Z-ranges returned from the decomposition stage, which includes merging of adjacent ranges. More ranges will give a more accurate approximation of the query region, and less false hits. However, an increase in the number of ranges also increases the overhead during query execution, which may cause a decrease in overall performance. The optimal number of ranges depends on the query, the data set, and the system. Generally, a beefier system will be able to handle more ranges.

The number of ranges is of great importance since it allows for the measurement of how well the best-first decomposition method adapts to non-uniform data compared to the breadth-first decomposition method by looking at the number of ranges needed by the two methods to avoid a similar amount of false positives.

Response Time

The response time is the primary metric used to evaluate the performance of our experiments and is defined as the time taken to execute a query. This includes the time spent on decomposition, fetching, and refinement. Ideally, the response should be as quick as possible and independent of the sizes of the index and data set. The response time should depend on the size of the result set of a query.

Decomposition Time

This decomposition time is the time spent on creating ranges for a query and should increase linearly with the number of ranges created. It is important that the additional time spent on decomposition not exceed the reduction in response time due to less refinement. Creating too many ranges can make the decomposition stage dominate the response time. Ideally, we want the decomposition time as low as possible while still achieving a low number of false positive hits.

False Discovery Rate

Another important metric is the percentage of false positive (FP) hits in the refinement stage compared to the total number of fetched objects during query processing. This

includes false positives during refinement and true positives (TP) in the result set. The false discovery rate (FDR) tells us the proportion of objects from the filter step that is removed in the refinement step.

$$FDR = \frac{FP}{FP + TP} \tag{5.1}$$

The false discovery rate indicates how well the ranges created by the decomposition method can approximate the query region given the data distribution. A low false discovery rate is desired as it represents less time spent on retrieving unwanted objects. The simplest solution to lower the false discovery rate is to create more ranges, but this comes at the cost of increased overhead and greater decomposition time. A goal of our proposed decomposition method is to achieve a low false discovery rate while limiting the number of ranges created.

5.6 Insertion Performance

High and scalable ingest performance is a necessity when dealing with the large dynamic datasets found in spatial and spatiotemporal applications. Our index structures are essentially B+tree indexes with some pre-calculations incorporated meaning that we should maintain the logarithmic scalability of the B+tree.

To compare the insertion performance of our indexes, we measure how long they take to construct compared to a normal B+tree.

Results

The insertion performance of the different index structures is presented in Figure 5.3. The data is inserted into the indexes sequentially, and they are then built entirely in-memory before committed to disk. Only the in-memory loading phase is shown in the figure. Each index is loaded with the entire data set. The spatiotemporal ($t-xyz$ and xyz) indexes have the slowest insertion speed while the B+tree index is fastest. The overall performance is high with an average of 105,000 points per second for the spatial (xy) index and an average of 80,700 and 83,500 points per second for the spatiotemporal indexes ($t-xyz$ and xyz, respectively). The insertion time for the spatiotemporal and spatial indexes are approximately 83% ($t-xyz$), 77% (xyz), and 41% higher (xy) compared to the B+tree. The insertion performance does not degrade for any of the indexes.

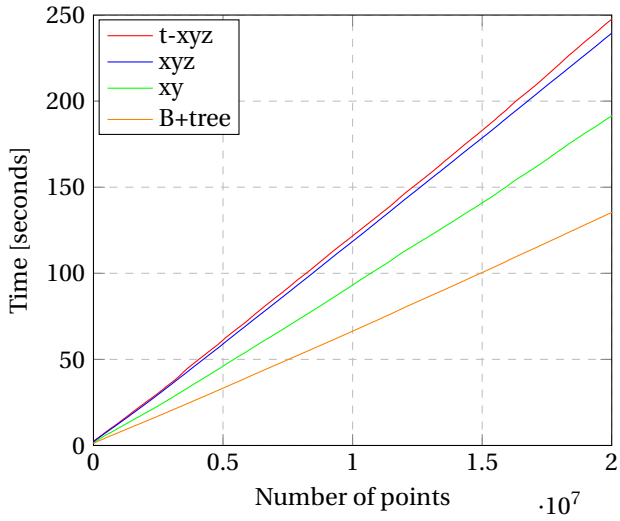


Figure 5.3: Insertion performance.

Index	Size
t-xyz	3.519 GB
xyz	3.320 GB
xy	2.891 GB

Table 5.2: The various index sizes.

Discussion

The worse insertion performance observed in the spatiotemporal indexes can be explained with the following: 1) The spatiotemporal encoding procedure is more computationally expensive as it includes an additional temporal encoding step to the spatial encoding step found in the spatial index. This includes extracting the timestamp from the data point and calculating the temporal offset. 2) The time complexity of the "magic bits" method used to encode the Z-address keys increase linearly with respect to the number of dimensions. 3) The 2D SFC component of the spatial index key is smaller with less serialization overhead compared to the 3D SFC component of the spatiotemporal index key (see Table 5.2).

Given the minor difference in insertion performance between the semi-integrated ($t-xyz$) and integrated (xyz) spatiotemporal indexes, it can be deemed that the overhead of serializing a larger index key is negligible compared to the effort of calculating the index key. The semi-integrated index key is significantly larger than the integrated index key as it contains an additional epoch component to the SFC address and the ID number.

5.7 Breadth-first Decomposition

This section tests the breadth-first decomposition method with a two-fold purpose of the experiments. First, we want to investigate the relationship of response time, decomposition time, and false discovery rate to the number of ranges created with the method. Second, given the results in this section, we want to determine the optimal parameter value for the maximum number of ranges for the breadth-first decomposition method in our test environments. The best-first decomposition method will be tested and compared against the tuned breadth-first decomposition method.

5.7.1 Maximum Number of Ranges

The effect of using a finer decomposition is obtained by measuring the average response time, decomposition time, and false discovery rate over the query set while gradually increasing the maximum number of ranges. The global minimum gives this optimal parameter value for the maximum number of ranges in the result graphs.

Results

Figure 5.4 shows how the average response time is affected by the maximum number of ranges for the different query sets. All the indexes show a clear inverse curvilinear relationship between the number of ranges and average response time where the benefits of a more fine-grained decomposition increase drastically to a certain point. The average response time for the spatial query set remains stable at 1100ms as the maximum number of ranges continuous to increase. However, for the spatiotemporal query set, the decomposition time starts to dominate the response time soon after the initial drop. When comparing the semi-integrated approach ($t-xyz$) with the integrated approach (xyz), the spatiotemporal queries are executed faster with the integrated approach than the semi-integrated approach (230ms versus 300ms, respectively).

Again, the number of range scans used to execute a region query is not the same as the (maximum) number of ranges returned from the breadth-first decomposition procedure. The ranges from the decomposition procedure are passed to the merging step, which merges adjacent ranges, before being executed as range scans. The range set is on average halved during the merging phase.

Figure 5.5 shows the relationship between the average false discovery rate and the maximum number of ranges. The average false discovery rate starts, as expected, to decrease when the number of ranges increases. The sudden drop in the false discovery rate reflects the same drop in response time in Figure 5.4. The average false discovery rate continues to decrease, although less drastically, throughout the range of the curve. This is contrary to the average response time, which starts to increase (or flat out for spatial queries) shortly after the knee of the curve.

The breadth-first decomposition method can achieve a much lower average false discovery rate for spatial queries compared to spatiotemporal queries given the same maximum number of ranges. Between the spatiotemporal indexes, the integrated approach appears to perform better.

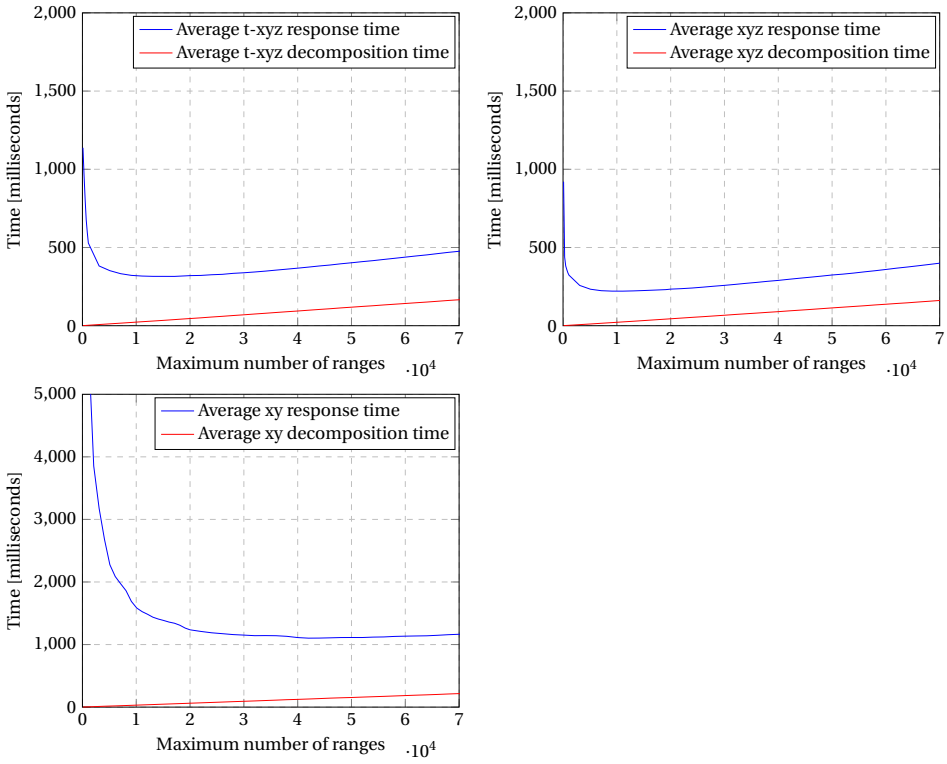


Figure 5.4: The effect of increasing the maximum number of ranges on the average response time with the breadth-first decomposition method.

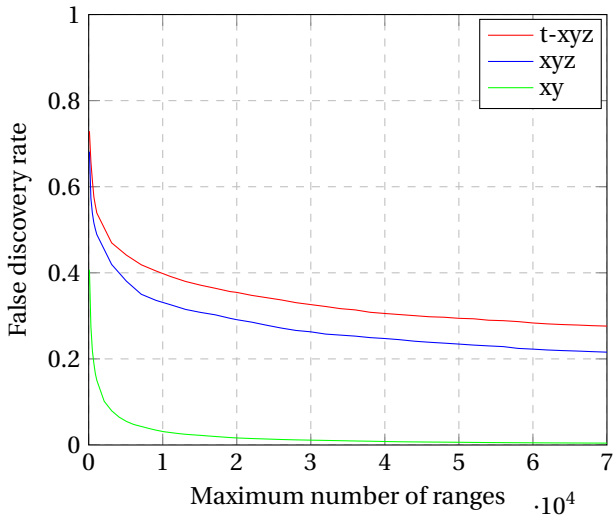


Figure 5.5: The effect of increasing the maximum number of ranges on the average false discovery rate with the breadth-first decomposition method.

Discussion

The worst average false discovery rate seen with spatiotemporal queries can be partly explained by the issue of scaling and the curse of dimensionality. First, the query region is decomposed more efficiently when the dimensions of the space-filling curve are of equal significance, i.e., the spatial and temporal predicates of the query fill the dimensions of the space-filling curve equally. The spatial predicates are of approximately equal length in our query set as the minimum bounding rectangles of the spatial query regions are approximately square. The sizes of the temporal predicates are, however, significantly different from the spatial predicates, which means that the minimum bounding cuboids of the spatiotemporal query regions will not be cube-shaped. Also, this issue cannot be trivially solved by fine-tuning the scaling of the curve dimensions as the optimal scaling depends on the query pattern. Second, decomposing a spatiotemporal query region with an octree accurately requires substantially more ranges than decomposing a spatial query region with a quadtree due to the increasing volume of space. The higher fanout of the octree will limit the precision of the decomposition. However, it is interesting to see that scaling used in our indexes causes the coarse hyper-quadrants created with the integrated approach to fit the spatiotemporal query regions better than the coarse hyper-quadrants created with the semi-integrated approach.

In addition to the poorer false discovery rate, the spatiotemporal queries are particularly sensitive towards the decomposition stage. After the initial drop in the false discovery rate, any benefit in reduced refinement time from its further reduction is outweighed by the increase in the decomposition time. The response time of the spatial queries remains, on the other hand, relatively stable as the maximum number of ranges increases. This behaviour may be explained first, and most importantly, by the spatial queries are of much higher volume than the spatiotemporal queries, which means that time spent on fetching and validating the points has a larger impact on the response time. The large difference in volume between the spatial and spatiotemporal queries can be observed by looking at the difference in average response time. Second, the decomposition procedure for spatiotemporal queries is on average 50% more expensive than for spatial queries. By increasing the number of dimensions, we also increase the average ratio between the number of disjoint and overlapping hyper-quadrants during the decomposition. The consequence is that more hyper-quadrants need to be checked to create a specific quantity of ranges for spatiotemporal queries than for spatial queries. Additionally, the cost of decoding Z-addresses during the decomposition increases with the number of dimensions.

5.8 Best-first Decomposition

In this section, we test and compare the best-first and breadth-first decomposition methods. Before we begin with the comparison, we construct a histogram for the best-first decomposition method. The datasets found in spatial and spatiotemporal applications are typically huge, and the construction and resource costs of the histogram may in many applications become prohibitively significant. Thus, we want to construct a histogram that provides sufficient accuracy with a minimum amount of resources. The resource cost of our histogram depends on the two parameters of the maximum prefix

length and sample size.

5.8.1 Maximum Prefix Length of the Histogram

As described in the previous chapter, the histogram is constructed incrementally with the index keys. The Z-address of a key is partitioned into prefixes of increasing length where each prefix is an entry in the hash table (i.e., a bucket in the histogram). The structure of the histogram can be thought of as a linear quadtree with interior nodes in addition to the leaf nodes, which together represent a sparse hyper-quadtree structure. An issue with this approach is that there will be many very small nodes (hyper-quadrants) at the bottom of the hyper-quadtree that is represented by the histogram. These nodes will usually contain very little data and are not worth expanding during query decomposition. Thus, resources can be saved by restricting the maximum prefix length of the hash table entries when constructing the histogram. This reduces the number of small buckets in the histogram and can be compared to restricting the depth of the hyper-quadtree. To determine the effect of adjusting the maximum prefix length of the histogram, we measure the performance over the query sets with histograms of increasing maximum prefix length. The prefix length is increased by N (number of dimensions) bits in each measurement, which is the same as increasing the height of the hyper-quadtree by one.

Results

Figure 5.6 presents how the performance of the best-first decomposition method is affected by the maximum prefix length of the histogram. The maximum prefix length is represented as the zoom level on the space-filling curve (maximum depth of the hyper-quadtree). The average response time remains relatively stable in the plots until a certain level where it starts to drop drastically. This applies particularly to the spatial query set, where the average response time drops from 30 seconds to 850 milliseconds. Contrary to the results from the previous experiments with breadth-first decomposition method, the semi-integrated spatiotemporal index achieves significantly better average response time here compared to the integrated spatiotemporal index (90ms versus 165ms for the integrated approach). The drop is also significantly steeper with the semi-integrated approach.

The average response time remains stable after the knee of curves. The decomposition time starts to increase for the semi-integrated spatiotemporal index while the response time remains flat. This indicates a balance between the time spent on less refinement and that of creating more ranges.

Figure 5.7 presents the relationship between the maximum prefix length of the histogram and the average false discovery rate over the query sets. The behaviour of the plots is, as expected, nearly identical to the behaviour seen in the plots for the average response time. Again, the average false discovery rate for the spatial query set drops more drastically than for the spatiotemporal query set. Comparing the semi-integrated approach with the integrated approach, we observe the drop in average false discovery rate for spatiotemporal queries is more significant for the semi-integrated approach, which is also reflected in the response time.

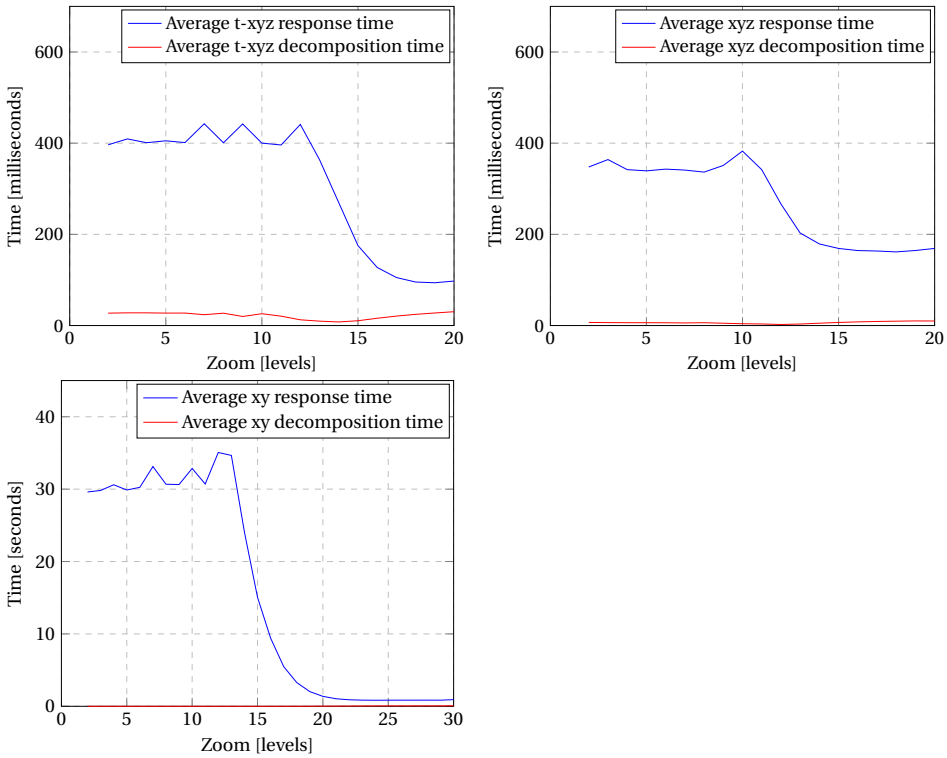


Figure 5.6: The effect of the maximum prefix length of the histogram on the average response time.

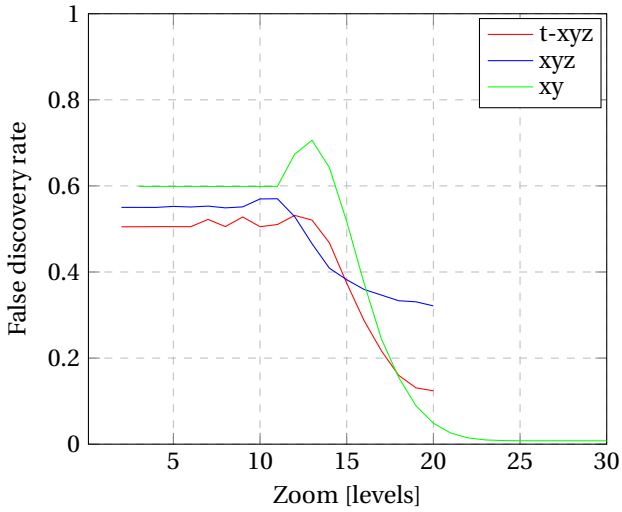


Figure 5.7: The effect of the maximum prefix length of the histogram on the average false discovery rate.

An interesting observation is a sudden bump in the false discovery rate right before the knee of the curve, which is more evident for the spatial query set. This is an artefact and is due to how to short circuit mechanism in the algorithm is implemented.

Overall, we reduce the construction cost of the histogram without degrading the performance by adjusting the maximum prefix length of the histogram according to the knees of the curves in Figure 5.6.

Discussion

The sudden drop in the false discovery rate and response time can be explained by the size of the minimum covering hyper-quadrant of the query region. Before the drop, the minimum covering hyper-quadrant is smaller than the smallest hyper-quadrant represented in the histogram. In other words, the Z-address prefix length of the minimum covering hyper-quadrant of the query region is larger than the maximum prefix length of the histogram. The consequence of this is that the search space will be considered empty by the histogram and the best-first decomposition method will start to short-circuit the decomposition procedure, which result is a coarse decomposition consisting of relatively few ranges that are likely to be passed to the expensive refinement step. This causes the high average response time and false discovery rate seen in Figure 5.6 and Figure 5.7. As soon as the maximum prefix length of the histogram surpasses the Z-address prefix length of the average minimum covering hyper-quadrant of the query set, the response time and the false discovery rate begin to drop drastically.

The effect of using different scaling on the temporal dimension is becoming more evident with the best-first decomposition method. The semi-integrated spatiotemporal index is superior to the integrated spatiotemporal index when the precision of the decomposition method (reachable depth of the hyper-quadtree) is not implicitly restricted due to branching. The limits on the decomposition precision imposed by the scaling of the space-filling curve dimensions are not evident with the breadth-first decomposition method as the method is unable to traverse the lower levels of the hyper-quadtree without overwhelming the system with ranges. The best-first decomposition method is, on the other hand, able to reach the lower levels of the hyper-quadtree and leverage the finer temporal resolution of the semi-integrated spatiotemporal index to create more precise ranges. In this experiment, the best-first decomposition method creates on average five times more ranges with the semi-integrated approach than with the integrated approach. However, it is interesting that the coarse temporal resolution of the integrated approach also prevents the best-first decomposition method from achieving a lower average false discovery rate than the breadth-first decomposition method due to rather aggressive short-circuiting.

5.8.2 Sampling Size of the Histogram

To determine how much sampling is sufficient for the histogram, we measure the average response time and false discovery rate over the query sets with a gradually increasing sampling size. There exists a variety of different sampling techniques, such as simple random sampling, systematic sampling, and stratified sampling, and the technique used in this experiment is the simple random sampling without replacement.

Results

Figures 5.8 and 5.9 show the how the average response time and average false discovery rate are affected by increasing sample sizes. The average response time converges shortly after the initial drop for all the indexes while the average false discovery rate continues to decrease more slowly. The following decrease in average false discovery rate after the initial drop is greatest for the semi-integrated spatiotemporal index. However, the decrease in response time becomes marginal due to the increase in decomposition time from creating more ranges. However, since the additional created ranges are likely to contain a substantial amount of data, the decomposition stage will never start to dominate the response time like with the breadth-first decomposition method, which tends to create many empty and coarse, intersecting ranges.

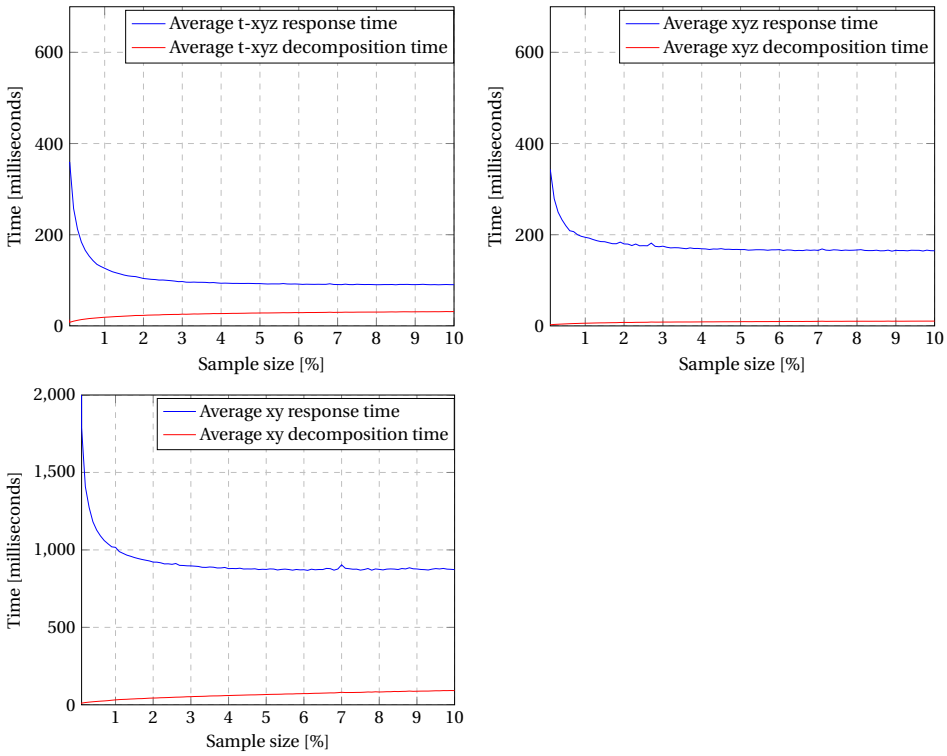


Figure 5.8: The effect of sampling size on the average response time.

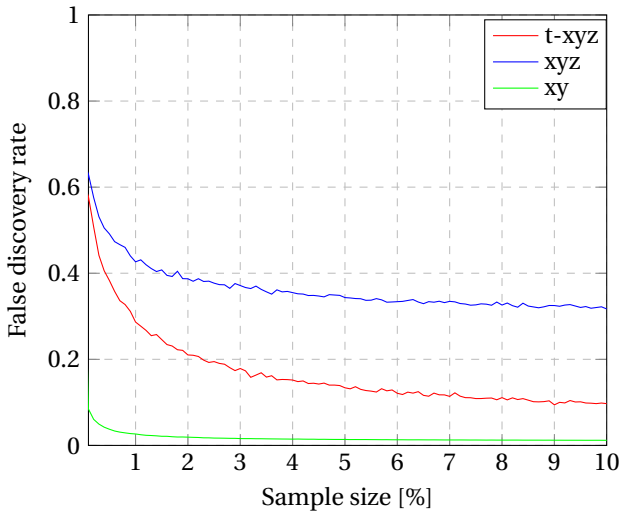


Figure 5.9: The effect of sampling size on the average false discovery rate.

Discussion

Based on these results, there appear to be very little or no benefits to increasing the sample size beyond the knees of the curves in Figure 5.8. This suggests we can reduce the constructing cost of the histogram significantly with sampling without having to sacrifice the performance.

5.8.3 Detailed Performance Comparison to the Baseline

In this section, we provide a more detailed comparison of the performance difference between the best-first decomposition method and the breadth-first decomposition method. The parameters used for the two decomposition methods are listed in Tables 5.3 and 5.4, which were selected based on the results from the previous experiments. For the histogram, we want minimize the memory cost without compromising the performance.

Histogram [index]	Zoom [levels]	Sample size [%]	Memory size [MB]
t-xyz	18	2	16.8
xyz	16	2	16.8
xy	23	2	16.8

Table 5.3: Parameter values used for the best-first decomposition method.

Index	Maximum number of ranges
t-xyz	10000
xyz	10000
xy	50000

Table 5.4: Parameter values used for the breadth-first decomposition method.

Results

The box plots of the response times for the two decomposition methods are shown in Figure 5.10. For the semi-integrated spatiotemporal index, we see the median line is lower for the best-first decomposition. The size of the first quartile is almost identical for the two decomposition methods, but the third quartile is much larger for the breadth-first decomposition method. The improvement in response time is significant according to the Wilcoxon signed rank test ($p < 0.05$).

For the integrated spatiotemporal index, the median of the best-first decomposition method is also lower than the median of breadth-first decomposition method. However, there is substantially more variation in the box plot of the best-first decomposition, which is seen by the larger first and third quartiles. However, the results are significant according to the Wilcoxon's test, so the best-first decomposition method is better.

For the spatial index, the median of the best-first decomposition method is lower than the median of the breadth-first decomposition method. Both the first and third quartiles are also larger for the best-first decomposition method. Again, the significance of the results is verified with Wilcoxon's test.

To conclude, according to the results of our experiments, the best-first decomposition method performs better than the breadth-first decomposition method for both spatial and spatiotemporal queries in our experiments. The largest improvement in performance is seen when processing spatiotemporal queries with the semi-integrated approach.

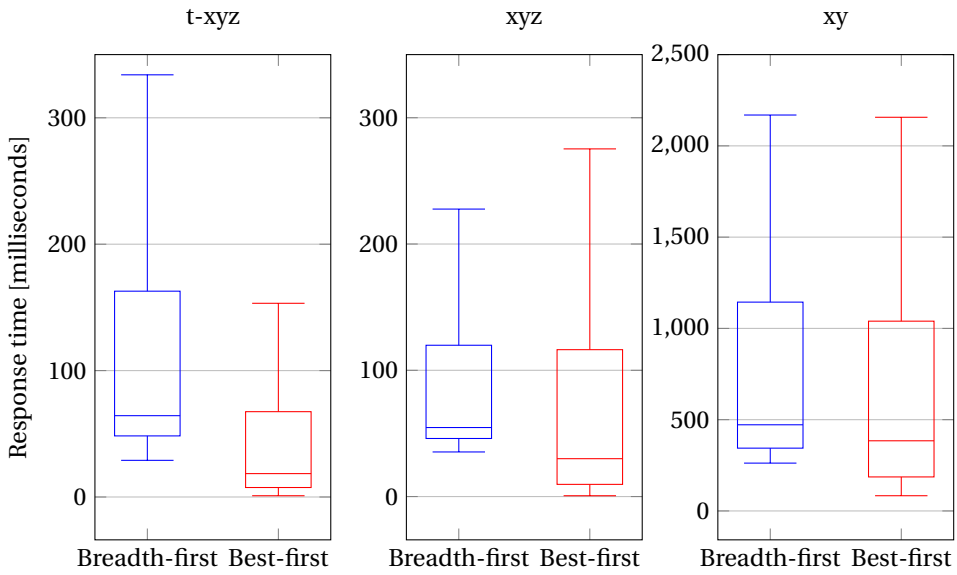


Figure 5.10: The performance of the breadth-first decomposition method and the best-first decomposition method.

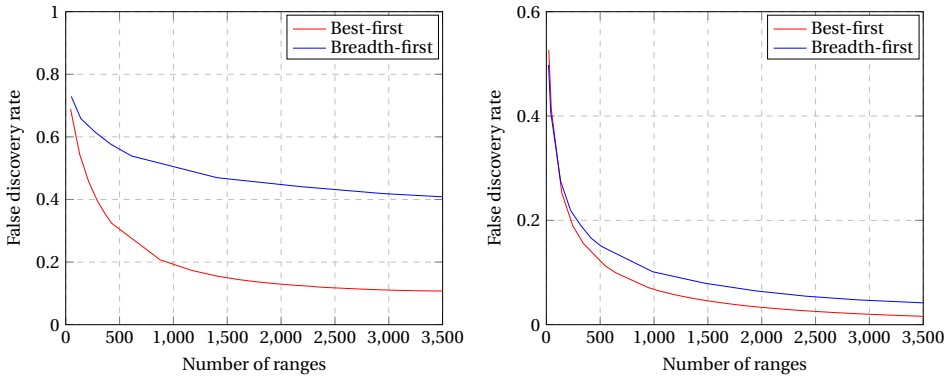
Discussion

The best-first decomposition was thought to perform better than the breath-first decomposition, and the results of our experiments confirm these initial expectations. The observed performance difference between the two approaches is due to two reasons. First, the breadth-first decomposition creates approximately the same amount of ranges for each query if the query region is not particularly small or an axis-aligned box. This means that the decomposition time of the breadth-first approach is relatively constant for similar query types (spatial or spatiotemporal) when the maximum number of ranges is fixed. The decomposition time of the best-first decomposition method depends, however, on the data distribution within the search region. The method stops the decomposition process when none of the remaining intersecting hyper-quadrants in the priority queue contains enough estimated data to be worth expanding. The ability to stop the decomposition process early enables us to execute low volume queries with extremely low response time (< 40ms). This is reflected by the lower whiskers in the box plots for the best-first decomposition method, which nearly reaches 0ms for spatiotemporal queries. The short-circuit mechanism is also seen to reduce the decomposition time for larger spatial queries by 150ms.

The second reason for the better performance of the best-first decomposition method is due to the capability of achieving a low false discovery rate with relatively few ranges. During the decomposition process, the best-first decomposition method will prioritize on partitioning intersecting hyper-quadrants that contain a large amount of data and will avoid expanding empty hyper-quadrants. The breadth-first decomposition will, on the other hand, not differentiate between intersecting hyper-quadrants in terms of how much data they are likely to contain and can waste a significant amount of time on creating ranges over empty regions. This behaviour is demonstrated by restricting the maximum number of ranges created by the best-first and breadth-first decomposition method. Figure 5.11 shows the relationship between the number of ranges created and the average false discovery rate for the two methods. The breadth-first decomposition method achieves an average false discovery rate of approximately 0.40 with 3,500 ranges for spatiotemporal queries and 0.04 with 3,500 ranges for spatial queries. The best-first decomposition method needs only 300 and 1,500 ranges to achieve similar average false discovery rate for spatiotemporal queries and spatial queries, respectively.[1]

The large difference in average false discovery rate between the two methods for spatiotemporal queries in Figure 5.11a indicates that the data is skewed within the spatiotemporal query regions. At the same time, the small difference in average false discovery rate between the two methods for spatial queries in Figure 5.11b indicates that the data is more uniformly distributed within the spatial query regions. This is likely to be the result of having the spatial query regions located within the densely populated areas of Beijing (see the heat map of the dataset in Figure 5.1).

[1] Both methods return approximately the same number of ranges for the spatiotemporal query set up to a maximum number of 2,000 ranges (this number is much higher for the spatial query set). Beyond that, the short-circuit mechanism of the best-first decomposition method begins to kick in and affect the average number of ranges returned.



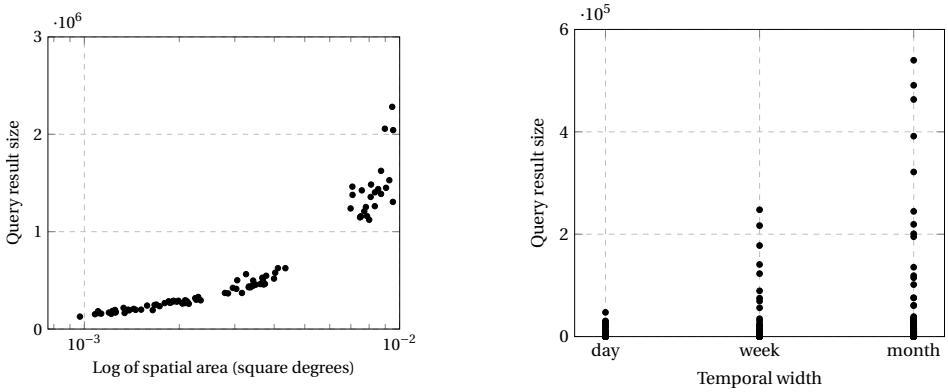
(a) Semi-integrated spatiotemporal index

(b) Spatial index

Figure 5.11: Average false discovery rate versus number of ranges created by the best-first decomposition method and the breadth-first decomposition method.

It should be mentioned again that the breadth-first decomposition method used in these experiments is static. A possible approach to making it more dynamic without relying on statistical data is to let the maximum number of ranges be defined as a function of the area of the query region and its shape. The primary assumption would be that the result size is proportional to the geometric size of the query, which is mostly correct for our experiments (see Figure 5.12) since all queries are located within the city boundaries of Beijing. However, this assumption is likely to break as soon we start to query less dense or rural areas. We may end up with a response time that will be dominated by the decomposition stage when the query regions consist of large remote areas, and by the refinement stage when the query regions enclose small and very dense areas. The potential of this approach will also likely be reduced for spatiotemporal queries, even when the assumption on the spatial distribution holds, as the data will likely be skewed in regards to the temporal dimension.

Earlier versions of GeoMesa, which used a now deprecated geohash index, did consider the geometrical properties of the query region during decomposition. The prior decomposition method used was based on a best-first decomposition strategy where the intersecting hyper-quadrant to be selected for further expansion is the one with the most area outside of the query region [4]. The specific maximum number of ranges to be created for a query was based on the ratio between the area of the query region and the area of the minimum bounding rectangle.



(a) Result size by query area.

(b) Result size by temporal width.

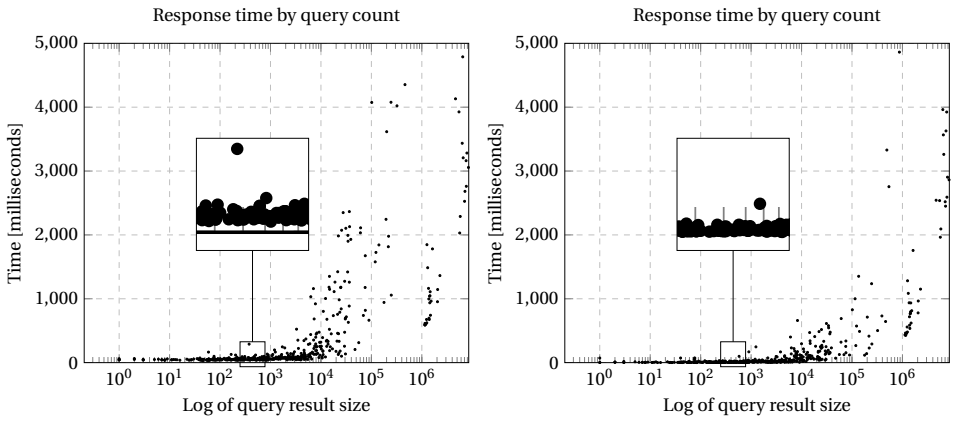
Figure 5.12: The relationship between the size of the query region and the result set size.

5.8.4 Query Size

This section examines the relationship between the response time and the result size.

Results

Figure 5.13 shows the relationship between the response time and the number of points in the result set. The upward trend shows that the response time depends mostly on the size of the result set. However, the variation in the response time is particularly large for medium and high volume queries (>10^4 points). This can be observed for both decomposition methods. However, the response time and variation are seen to be generally lower with the best-first decomposition method. By zooming in on the graph, we observe the difference between the lower bounds for the best-first decomposition (> 0ms) method and the breadth-first decomposition (> 40ms).



(a) Breadth-first decomposition

(b) Best-first decomposition

Figure 5.13: The relationship between the result size and the response time.

Discussion

The variation in response time seen between similarly sized queries is the result of having a CPU-bound workload (by using memory-mapped files and having enough physical memory to copy the entire dataset into memory). The point-in-polygon tests make the refinement stage very expensive, and a reduction in false discovery rate can substantially decrease the response time when the amount of data to be returned is large. Thus, queries of relatively high volume (10^6 points) can be executed with a relatively low response time (< 500ms) if the false discovery rate is low and the decomposition is fast. The lower variation in response time between medium volume queries in Figure 5.13b compared to Figure 5.13a are due to spatiotemporal queries, which are executed with a significantly lower average false discovery rate with the best-first decomposition method than with the breadth-first decomposition method.

5.8.5 Polygon Complexity

This section measures the impact of polygon complexity on the false discovery rate. The polygon complexity measure is a quantitative description of the polygon shape and describes the degree of difference between a regular shaped polygon, which is taken from [21]. The advantage of this measure over other complexity measures, like fractal dimension [27], is that it takes the global shape of the polygons into consideration. The complexity model consists of three different parameters: frequency of local vibration, the amplitude of the local variation, and deviation from the convex hull. The frequency of local vibration ($freq$) describes the variability of the boundary and the measure is based on the ratio between the number of polygon notches (concave vertices) and the total number of polygon vertices.

$$freq(p) = 16 \times (notches_{norm}(p))^4 - 8 \times (notches_{norm}(p) - 0.5)^2 + 1 \tag{5.2}$$

$$notches_{norm}(p) = \frac{notches(p)}{vertices(p) - 3} \tag{5.3}$$

The amplitude of the local vibration ($ampl$) describes the intensity of the local vibration and is measured as the relative length difference between the boundary of the polygon and the boundary of its convex hull (i.e., its smallest enclosing convex polygon).

$$ampl(p) = \frac{boundary(p) - boundary(p_{covex\ hull})}{boundary(p)} \tag{5.4}$$

Deviation from the convex hull ($conv$) describes the global complexity of the polygon and is measured as the relative difference between the area of the polygon and the area of its convex hull.

$$conv(p) = \frac{area(p_{covex\ hull}) - area(p)}{area(p_{covex\ hull})} \tag{5.5}$$

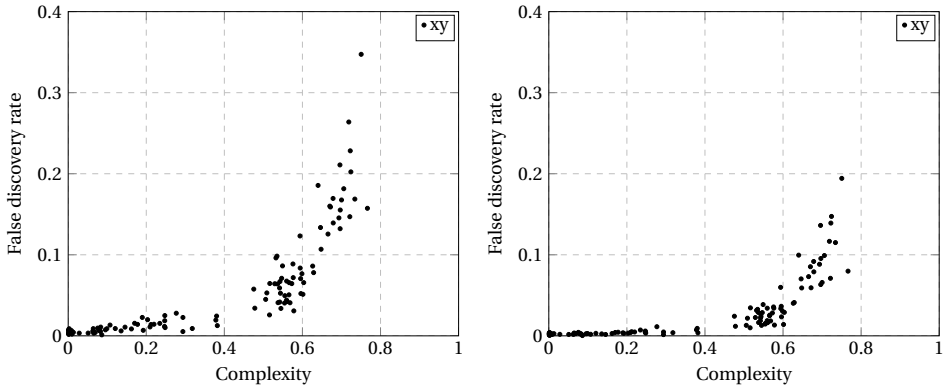
The three parameters $freq, ampl$ and $conv$ are combined into a single measure of complexity ($comp$) with the range of $[0,1]$, such that

$$comp(p) = 0.8 \times ampl(p) \times freq(p) + 0.2 \times conv(p) \tag{5.6}$$

Results

Figure 5.14 shows how the false discovery rate varies with the measured polygon complexity of spatial query regions when the decomposition methods are restricted to create a maximum number of 5,000 ranges. For clarity reasons, we omit spatiotemporal queries as the impact of the temporal dimension on the false discovery rate tends to overshadow the impact of spatial polygon complexity on the false discovery rate.

Both methods show nearly exponential growth in the false discovery rate as the polygon complexity increases. However, the polygon complexity affects the false discovery rate of the breadth-first decomposition method nearly twice as much compared to the best-first decomposition method when the number of ranges is restricted.



(a) Breadth-first decomposition **(b)** Best-first decomposition

Figure 5.14: The relationship between the measured polygon complexity and the false discovery rate.

Discussion

The worse false discovery rate seen with breadth-first decomposition method was expected as it becomes more difficult to fit larger quadrants within complex polygons. Additionally, the increasing difference in area between the polygon and its convex hull will increase the amount of the surplus space within the intersecting quadrants when the precision of the decomposition is restricted. The best-first decomposition will be more robust against complex polygons as the quadrant precision of the method is not severely restricted due to branching.

However, the polygon complexity has still a large impact on the false discovery rate of the best-first decomposition method in this experiment. As mentioned earlier, the data appears to be relatively uniformly distributed within the spatial query regions. This causes the best-first decomposition method to decompose the spatial queries in a more breadth-first manner and must, therefore, generate a large number of ranges to achieve a low false discovery rate for polygons of high complexity. The difference in false discovery rate between the two methods is expected to be larger if most of the retrieved

data were located along smaller portions of the polygon boundary.

It is worth noting that the model we have used to measure polygon complexity has the property of reflecting the cost of answering a point-in-polygon test when a pre-computed spatial-hierarchy index is created for the polygon (in the paper, a TR*-tree [32] is used, which usually provides logarithmic PIP performance). This means that the refinement process is likely to become more expensive as the polygon complexity increases, which gives the best-first decomposition method an additional performance advantage over the breadth-first decomposition method when querying highly irregular shaped polygons.

5.9 Parallelization

In the experiments performed so far, we used only a single thread to execute queries. We now look at the performance of the best-first decomposition method and how it is affected by incorporating parallelization. We are expecting to see a significant speed up as our system is CPU bound. The ideal case is a linear performance increase by the number of threads. However, there exist multiple factors that can have a large impact on scalabilities, such as memory bandwidth saturation and false sharing of cache lines. Also, several stages of the query processing are not parallelized, such as merging adjacent ranges and fetching objects from the result set, which will affect the potential performance increase. The experiment performed here focuses on query execution and not on indexing as the concurrency potential of the indexing procedure is likely bound to the implementation of the underlying B+tree.

Our hardware restricts us to use a maximum number of four threads. The test setup is identical to previous experiments, and the queries are executed as before one at a time, but a single query is now executed in parallel using multiple threads.

Results

The box plots in Figures 5.15 and 5.16 examine the effect of using multiple threads for query processing. The average speedup is shown in Table 5.5 where the gain from using i threads over a single thread is given by $S_i = \frac{T_1}{T_i}$.

As expected, parallelization improves both the overall response and decomposition times. There is little improvement in the median of the box plots in Figures 5.15 and 5.16 compared to the average speed up in Table 5.5. This is due to the largest speedup is found in large volume queries, which can be observed by the reduction in the upper third quartile and the upper whisker in the box plots when increasing the number of threads.

The integrated spatiotemporal index has the greatest overall average speedup (S_4 = 2.43) while the spatial index has the lowest overall average speedup (S_4 = 1.25). On the other hand, the integrated spatiotemporal index has the worst average decomposition speedup (S_4 = 1.56). The average decomposition time with four threads are roughly cut

in half compared to one thread with the spatial (S_4 = 2.03) and semi-integrated spatiotemporal (S_4 = 2.04) indexes .

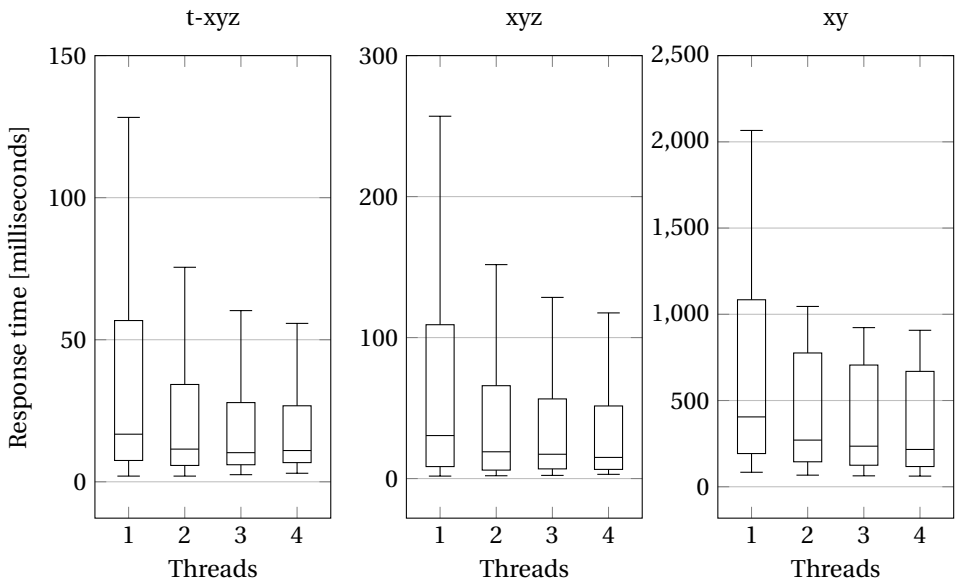


Figure 5.15: The response time of the best-first decomposition method with increasing number of threads.

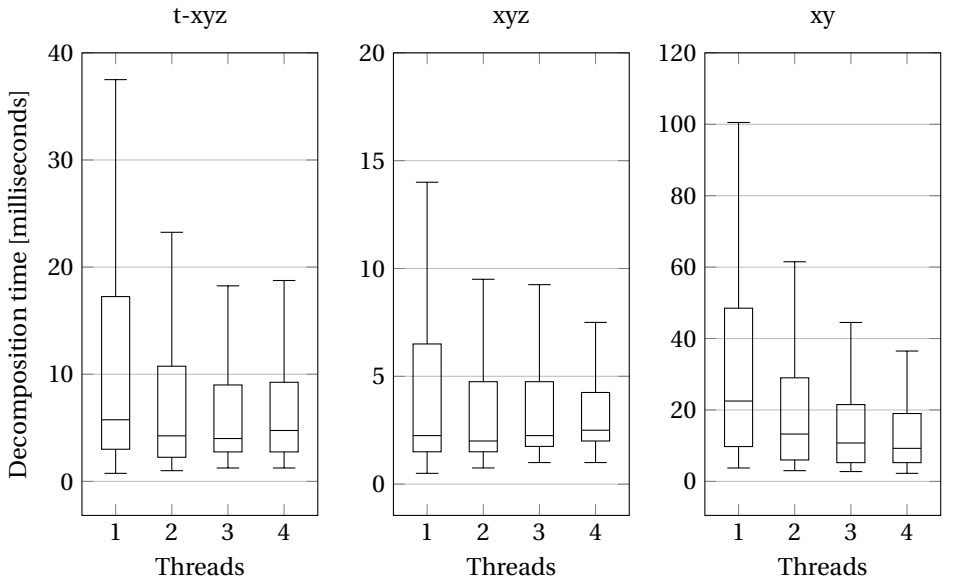


Figure 5.16: The decomposition time of best-first decomposition method with increasing number of threads.

Table 5.5: Average speedup by using multiple threads.

(a) Response Time

S_i	t-xyz	xyz	xy
S_1	1.00	1.00	1.00
S_2	1.57	1.69	1.15
S_3	1.94	2.17	1.19
S_4	2.10	2.43	1.25

(b) Decomposition Time

S_i	t-xyz	xyz	xy
S_1	1.0	1.0	1.0
S_2	1.48	1.44	1.56
S_3	1.87	1.46	1.79
S_4	2.04	1.56	2.03

Discussion

The largest contributing factor to the overall speedup is the number of false positives. Spatiotemporal queries executed with the integrated approach experiences the largest overall performance gain because of the parallelized refinement stage. The integrated approach has the highest average false discovery rate, which explains the improved performance from parallelizing the refinement stage. Spatial queries, on the other hand, have a very precise decomposition with a very low average false discovery rate, which means the refinement stage has less of an effect on the overall response time (fetching objects from the result set is not parallelized). However, the spatial index and semi-integrated spatiotemporal index benefit more than the integrated spatiotemporal index from parallelizing the decomposition stage as they tend to create significantly more ranges.

Parallelization benefits also high volume queries more than low volume queries due to less impact from the overhead involved with creating and managing multiple threads. This includes the job of performing a shallow decomposition and distributing the intersecting hyper-quadrants to the threads, and perform additional decompositions in parallel and merge the results.

Even though the response time decreases significantly by parallelization in this experiment, it is the speedup of decomposition procedure that is of most interest. First, time spent on decomposing the query into subqueries was, as in several cases in the related works [31], [20], and [25], found to degrade the response time. Second, the refinement stage would naturally be highly parallelized in NoSQL systems that support server-side filtering. Both MD-HBase and GeoMesa leverage push-down predicates to parallelize the refinement stage through the notion of server-side iterators and co-processors in Accumulo and HBase, respectively. By parallelizing the decomposition procedure, we reduce the latency between issuing a region query and receiving the first results.

5.10 Validation

Our space-filling curve indexes rely on lossy encoding and are not designed to encode and decode coordinate positions perfectly, but instead to efficiently narrow down the search space into a set of one-dimensional range queries. The consequence of using a

lossy encoding scheme with our decomposition method is that we may lose some data that lies at the boundary of the query region. The expected loss of information depends on the curve precision and will decrease when the number of bits used to encode the Z-addresses increases. The spatiotemporal index is expected to be more inaccurate than the spatial index as it uses 21 bits per dimension, which gives it a precision of approximately 100 meters, compared to 31 bits per dimension, which gives the spatial index a precision of roughly 10 centimetres.

Index	Total points returned	Points missed	Average loss
t-xyz	7,954,933	535	6.7×10^{-5} %
xyz	7,955,447	21	2.6×10^{-6} %
xy	246,556,706	17	6.9×10^{-8} %

Table 5.6: The number answer points returned from the best-first decomposition method compared to true number of answer points.

Table 5.6 shows the average retrieval loss for the different indexes when using best-first decomposition method. The spatial index is in the order of magnitudes more accurate than the spatiotemporal indexes. Between the spatiotemporal indexes, the semi-integrated approach displays significantly larger retrieval loss than the integrated approach.

A more fine-grained decomposition is likely to have a higher retrieval loss compared to a coarse-grained decomposition as the coarser decomposition will include a larger buffer around the query region to be processed in the refinement stage. The semi-integrated approach experiences a larger retrieval loss than the integrated approach as it decomposes the query region more precisely.

The problem of retrieval loss can be reduced by using larger Z-addresses to index the points or by restricting the maximum allowable range-precision and include an additional buffer to the query region during decomposition. These approaches will, however, significantly affect the performance so that the optimal solution will depend on the specific use-case.

Chapter 6

Conclusion and Further Work

This chapter contains the conclusion of our work and presents ideas that could be worth investigating further.

6.1 Conclusion

An increasingly popular approach to support large spatial and spatiotemporal data in key-value NoSQL systems is to use locality-preserving space-filling curves to linearize multidimensional data. Region queries are executed efficiently in these systems by decomposing them into sets of linear ranges scans supported by the underlying storage model. In this thesis, we have investigated the common approaches of decomposing region queries into linear space-filling curve ranges in NoSQL systems. Based on this research, we have proposed a quadtree-based decomposition method that decomposes spatial and spatiotemporal region queries in a best-first manner by using a prefix frequency histogram over the index keys as a cost function. The method has been implemented in a simple database prototype and evaluated with real-world data.

To evaluate the performance of our method, we have implemented and used the popular quadtree-based breadth-first decomposition method as a baseline. In the performed experiments, our method was able to outperform the baseline in terms of response time and variability for both spatial and spatiotemporal queries. The observed difference in performance between the baseline and our proposed method can mainly be attributed to the ability to adjust the decomposition dynamically to the data distribution and density, and implicitly to the shape of the query region. For instance, the capability to short-circuit the decomposition early enabled our method to execute low volume queries with significantly lower response time than the baseline method, which creates approximately the same number of ranges for each query. This data-driven adaptability makes our method especially robust against different query patterns and data sets.

We experienced that our method was more effective against spatiotemporal queries

than spatial queries compared with our baseline. This was attributed to several factors: decomposing a spatiotemporal query accurately with the baseline method requires on average substantially more ranges than decomposing a spatial query due to the additional temporal dimension, which greatly increases the volume of the query region. In addition, the small difference in false discovery rate between our method and the baseline when executing spatial queries with a similar amount of ranges indicated a relatively uniform data distribution within the spatial query regions, which was not the ideal when it came to demonstrating the efficiency of our method. However, in general, our method has demonstrated to be capable of reducing the number of ranges needed to avoid false hits and, thus, reduce the cost associated with creating and issuing range scans.

An interesting challenge we ran into during the thesis was how the different ways of integrating time into the space-filling curve affect the query performance. By integrating the entire time range of the data set into a single curve, the performance of our method degraded significantly due to a coarse temporal resolution. However, better results were found when we partitioned time into year-long blocks and having a separate Z-order curve for each block, which resulted in a fixed high temporal resolution and more precise decompositions. The cost of this approach is storage and insertion overhead. However, the additional overhead is well-worth the improvement in query performance and the fact that it allows us to avoid any overflow issues when dealing with dynamic datasets.

Overall, we showed in this thesis that the performance of spatial and spatiotemporal region queries against space-filling curve indexes can be improved by leveraging statistical data during the decomposition process. The proposed decomposition method is yet to be tested in a distributed context with a NoSQL system. Even though modern NoSQL systems can mitigate the impact of pruning false positives by using server-side filters, we believe our method will allow us to reduce the overall load on the cluster and response time while still retaining high insertion performance and scalability. The cost of maintaining the histogram can also be reduced significantly with negligible performance cost by taking measures such as sampling and removing small-sized buckets.

6.2 Further Work

In this section, we present some further aspects worth investigating.

- There exist space-filling curves that preserve locality better than the Z-order curve. An example is the Hilbert curve. By using a Hilbert curve instead of a Z-order curve, we would likely decrease the false discovery rate further. However, the additional overhead from encoding and decoding more computationally complex Hilbert addresses might outweigh the benefit of a more accurate decomposition.
- In addition to statistical data, we could also include the current system context when determining the next hyper-quadrant to expand during the decomposition process. When dealing with very large datasets, it is likely that only portions of the index will fit into main memory. The decomposition procedure can then min-

imize expensive disk access due to false positives by decomposing portions of the query region that are not already in memory more precisely.

- The relatively low bit precision used to encode the Z-addresses in this thesis introduced some minor retrieval loss. For use-cases that depend more on exact results than performance, it might be worth the additional overhead of using a format with unrestricted precision, e.g., `byte[]`, to store the Z-address keys.

- By investigating how fast the underlying distribution of dynamic spatial and spatiotemporal datasets changes, we can determine the need for keeping the histogram up-to-date. If the distribution were to remain relatively stable over time we may decrease the update frequency of the histogram without affecting the performance. We may also try to use more complex techniques to estimate the data distribution, e.g., with kernel density estimation.

- Space-filling curve indexes have an inherent lack of support for efficient nearest-neighbor queries. For instance, a k-nearest-neighbor query is executed in GeoMesa by iteratively expanding and processing the target space in a spiral until k neighbors are found or to confirm the current found points are actually the nearest neighbors. A problem with this approach is that the granularity of the search (how many meters outwards the target space is to be expanded per iteration) must be guessed by the client beforehand. By using a histogram similar to the one used in this thesis, we could automatically estimate the ideal expansion ratio during query processing.

- Our prototype is relatively simple and does not support distributed processing. To support and test our proposed method against very large spatial and spatiotemporal datasets, it would be necessary to implement it on top of a NoSQL system.

- If we were to implement the method on top of the NoSQL store Accumulo, we would have to execute two separate `BatchScanner` jobs — one for intersecting ranges and one for contained ranges. Because the additional overhead of executing two scanner jobs instead of one might degrade the performance of low volume queries, it would be interesting to see if it is worth using the histogram to determine if a query is likely to be a low volume query. All ranges generated from decomposing a low volume query can then be marked as intersecting and only a single scanner job is executed.

- The technique used in this thesis can trivially be generalized to support N-dimensional range queries. Thus, it would interesting to see if our proposed method can be used efficiently in applications that handle data with more than three dimensions.

Bibliography

[1] Apache accumulo 1.6.6. https://accumulo.apache.org/release/accumulo-1.6.6/. (Accessed on 05/11/2018).

[2] B trees. http://www.cs.gordon.edu/courses/cs321/lectures/Btrees. (Accessed on 05/08/2018).

[3] fastutil. http://fastutil.di.unimi.it/. (Accessed on 05/19/2018).

[4] geomesa/GeohashUtils.scala · locationtech/geomesa · GitHub. https://github.com/locationtech/geomesa/blob/geomesa_2.11-1.3.0-m0/geomesa-utils/src/main/scala/org/locationtech/geomesa/utils/geohash/GeohashUtils.scala#. (Accessed on 05/20/2018).

[5] GitHub - locationtech/sfcurve: LocationTech SFCurve is a Scala library for the creation, transformation, and querying of space-filling curves. https://github.com/locationtech/sfcurve. (Accessed on 05/19/2018).

[6] Google support - view traffic. https://support.google.com/maps/answer/3092439?visit_id=1-636463496461694935-1097629522&rd=1. (Accessed on 06/06/2018).

[7] Home · GeoMesa. http://www.geomesa.org/. (Accessed on 05/05/2018).

[8] Introduction — GeoMesa 2.0.0 Manuals. http://www.geomesa.org/documentation/user/introduction.html#what-is-geomesa. (Accessed on 05/11/2018).

[9] JTS Topology Suite - OSGeo. https://www.osgeo.org/projects/jts/. (Accessed on 06/09/2018).

[10] MapDB. http://www.mapdb.org/. (Accessed on 05/18/2018).

[11] Morton encoding/decoding through bit interleaving: Implementations – Jeroen Baert's Blog. http://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/. (Accessed on 05/19/2018).

[12] postgres/README· postgres/postgres · GitHub. https://github.com/postgres/postgres/blob/master/src/backend/access/nbtree/README. (Accessed on 05/18/2018).

[13] R-trees. http://cglab.ca/~cdillaba/comp5409_project/R_Trees.html. (Accessed on 06/06/2018).

[14] Snap Map FAQ. https://support.snapchat.com/en-US/a/snap-map-faq. (Accessed on 05/16/2018).

[15] View frustum culling of point clouds using octrees in SimpleEXR - Benjamin Weißer. http://benjaminweisser.com/blog/20170328-view-frustum-culling-of-point-clouds-using-octrees-in-simpleexr. (Accessed on 06/06/2018).

[16] Who is using MapDB - MapDB. http://www.mapdb.org/success/. (Accessed on 05/18/2018).

[17] Ashraf Aboulnaga and Walid G Aref. Window query processing in linear quadtrees. *Distributed and Parallel Databases*, 10(2):111–126, 2001.

[18] Rudolf Bayer. The universal b-tree for multidimensional indexing. 1996.

[19] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. ACM, 1990.

[20] Mohamed Ben Brahim, Wassim Drira, Fethi Filali, and Noureddine Hamdi. Spatial data extension for cassandra nosql database. *Journal of Big Data*, 3(1):11, 2016.

[21] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Alexander Braun. Measuring the complexity of polygonal objects. In *ACM-GIS*, page 109, 1995.

[22] Rick Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[24] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[25] Andrew Hulbert, Thomas Kunicki, James N Hughes, Anthony D Fox, and Christopher N Eichelberger. An experimental study of big spatial data systems. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 2664–2671. IEEE, 2016.

[26] Philip L Lehman et al. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, 1981.

[27] Benoit B Mandelbrot. *The fractal geometry of nature*, volume 173. WH freeman New York, 1983.

[28] Mohamed F Mokbel and Walid G Aref. On query processing and optimality using spectral locality-preserving mappings. In *International Symposium on Spatial and Temporal Databases*, pages 102–121. Springer, 2003.

[29] Mohamed F Mokbel, Walid G Aref, and Ibrahim Kamel. Analysis of multi-dimensional space-filling curves. *GeoInformatica*, 7(3):179–209, 2003.

[30] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.

[31] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 7–16. IEEE, 2011.

[32] Ralf Schneider and Hans-Peter Kriegel. The tr*-tree: A new representation of polygonal objects supporting spatial queries and operations. In *Workshop on Computational Geometry*, pages 249–263. Springer, 1991.

[33] Peter van Oosterom and Tom Vijlbrief. The spatial location code. In *Proceedings of the 7th international symposium on spatial data handling, Delft, The Netherlands*, 1996.

[34] Yu Zheng, Hao Fu, Xing Xie, Wei-Ying Ma, and Quannan Li. Geolife gps trajectory dataset-user guide, july 2011. *URL: https://www. microsoft. com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide*.

[35] Yu Zheng, Quannan Li, Yukun Chen, Xing Xie, and Wei-Ying Ma. Understanding mobility based on gps data. In *Proceedings of the 10th international conference on Ubiquitous computing*, pages 312–321. ACM, 2008.

[36] Yu Zheng, Lizhu Zhang, Xing Xie, and Wei-Ying Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proceedings of the 18th international conference on World wide web*, pages 791–800. ACM, 2009.

[37] Kathryn Zickuhr. Three-quarters of smartphone owners use location-based services. *Pew Internet & American Life Project*, 2012.