TU WIEN Informatics

SPX

Ein vielseitiges erweiterbares räumliches Indexierungsframework

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Markus Töpfer, BSc
Matrikelnummer 00525762

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Math. Dr.techn. Katja Bühler, VRVis
Dipl.-Ing. Dr.techn. Florian Ganglberger, VRVis

Wien, 16. September 2024

Markus Töpfer	Eduard Gröller

TU WIEN Informatics

SPX

A Versatile Extensible Spatial Indexing Framework

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Markus Töpfer, BSc
Registration Number 00525762

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Math. Dr.techn. Katja Bühler, VRVis
Dipl.-Ing. Dr.techn. Florian Ganglberger, VRVis

Vienna, September 16, 2024

Markus Töpfer | Eduard Gröller

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Markus Töpfer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 16. September 2024

Markus Töpfer

Acknowledgements

I am profoundly grateful to the VRVis Research Center for giving me many years of interesting topics to work on and enabling me to hone the necessary skills for my future. Especially, I want to mention Katja Bühler and Florian Ganglberger, who are responsible for providing a foundation for this thesis.

Also, I want to thank my supervisor Eduard Gröller for his patience and guidance.

On a personal note, I am eternally grateful to my family for their love, patience, and encouragement. To my parents, Edith and Dieter, thank you for your unwavering belief in me, your endless support, and your understanding during the heights and lows of this journey.

Lastly, I would like to thank everyone who contributed to this thesis in any capacity, whether directly or indirectly. Your contributions, no matter how small, have been invaluable.

Thank you all.

This work was enabled by the Competence Centre VRVis and funded by the Austrian Science Fund (FWF) project number I4836. VRVis is funded by BMK, BMAW, Styria, SFG, Tyrol, and Vienna Business Agency in the scope of COMET - Competence Centers for Excellent Technologies (879730) which is managed by FFG.

Kurzfassung

Moderne genetische Markierungstechniken, kombiniert mit fortschrittlichen Mikroskop- und Zellextraktionsmethoden, ermöglichen es Neurobiologen, umfangreiche digitalisierte und ko-registrierte Probensammlungen von Gehirnen verschiedener Spezies zu erstellen.

Diese Sammlungen dienen als wertvolle Ressourcen zur Untersuchung neuronaler Strukturen, funktionaler Regionen und neurologischer Verbindungen im Gehirn. Durch die Entnahme einzelner Zellen aus verschiedenen Bereichen des Tiergehirns können Wissenschaftler Verteilungen von Zelltypen und Genexpressionen untersuchen.

Das Erkunden dieser umfangreichen Sammlungen, die aus volumetrischen Bildern, segmentierten Strukturen und Genexpressionsdaten bestehen, stellt jedoch eine erhebliche Herausforderung in der Neurowissenschaft dar. Ein effizientes Durchsuchen der Sammlungsdaten in abgefragten Bereichen, der performante Zugriff auf Daten und eine schnelle Verarbeitung der Sammlungsdaten sind für Forscher von entscheidender Bedeutung.

In dieser Arbeit präsentiere ich einen flexiblen und erweiterbaren Ansatz zur räumlichen Indizierung und Speicherung von volumetrischen Voxel- und regionsbasierten Daten. Dieser Ansatz ermöglicht einen effizienten Zugang zu den Daten und eine performante Verarbeitung. Das vorgestellte Rahmenwerk unterstützt verschiedene Datensätze, die indizierung verschiedener Datentypen, und integriert einen Daten-Partitionsmechanismus, um mehrere Datenrepräsentationen oder zeitabhängige Daten innerhalb einer einzigen Datenstruktur zu verwalten. Standardisierte Schnittstellen ermöglichen eine einfache Implementierung neuer Datenabstraktionen, Abfragearten, Indizierungsstrategien und neuer Speichermöglichkeiten.

Diese Arbeit bietet einen Überblick über konzeptionelle Ideen, Implementierungsdetails, aktuelle Datenabstraktionen und Abfragetypen. Das System wurde hinsichtlich Leistung und Skalierbarkeit in seinen aktuellen Anwendungsfällen bewertet. Eine kurze Einführung in drei Anwendungen – BrainBaseWeb, BrainTrawler und BrainTrawler Lite – veranschaulicht die Nutzung dieses Rahmenwerks.

Abstract

The integration of modern genetic techniques, advanced volumetric imaging methods, and single-cell extraction methods has empowered neurobiologists to create extensive digitized and coregistered specimen sample collections. These collections serve as valuable resources for studying neuronal structures, functional compartments, and neurological connections within the brain. By sampling single cells from various locations within the animal brain, scientists can investigate cell type distributions and gene expressions.

However, the exploration of these vast collections, which include volumetric images, segmented structures, and gene expression data, poses a significant challenge in neuroscience. Efficient access to specific regions of interest in all images, derivative processed data, cell samples, and metadata is essential for researchers.

In this thesis, I present a flexible and extensible approach to spatially index and store volumetric grid data and region-based data, enabling efficient access and providing a streamlined method for implementing new data abstractions, query types, and indexing strategies. The framework supports different datasets, the encoding of neurological structural types, and incorporates a layering mechanism to handle multiple data representations or time-dependent data within a single data structure. Standardized interfaces are defined for loading voxel and region data, preprocessing them, creating data abstractions, and implementing new query types. The data storage is managed using a storage engine approach, allowing users to leverage different storage mechanisms or introduce their own.

This thesis provides an overview of conceptual ideas, implementation details, current data abstractions, and query types. The system was evaluated in terms of performance and scalability in its current use cases. A short introduction to three applications, BrainBaseWeb, BrainTrawler, and BrainTrawler Lite, exemplifies the usage of this framework.

Contents

CHAPTER 1

Introduction

Recent developments in neuroscience, combined with modern imaging technologies, have led to the generation of vast collections of volumetric brain data. Single-cell extraction mechanisms and RNA sequencing techniques have enabled neuroscientists to explore the brain's cellular composition and create gene expression profiles for samples taken from different brain regions.

Researchers rely on high-resolution imaging techniques to explore intricate structures within biological specimens. The distribution of cell types and properties, as well as the expressiveness of genes in different regions of the brain, are essential for understanding the brain's functionality. Storing, retrieving, and querying these collections of volumetric data and single-cell gene expression data in a spatial context pose significant challenges. Traditional database systems and indexing techniques often fall short when dealing with the complexities of large amounts of three-dimensional spatial data, leading to inefficiencies in data access and query performance.

This thesis addresses these challenges by presenting SPX, a versatile and extensible spatial indexing (*SPX*) framework designed to optimize the storage, retrieval, and processing of volumetric data in two forms: grid-based data and region-based data. Building on proven concepts for grid-based data from predecessor projects that leverage space-filling curves, my contribution lies in the integration of the region-based data concept with support for multiple data samples per region, the introduction of abstraction layers and interfaces to easily integrate new data representations, novel queries, different indexing strategies, and use-case optimized storage mechanisms. A novel data layer approach enables the storage of several data entries for the same indexed sample within the same index structure.

The SPX framework not only supports efficient access to diverse data types but also facilitates the integration of new data abstractions, providing a robust solution for the ever-growing demands of spatial data management in computer science.

SPX is a core part of the Brain* framework [85], which is used in the Larval Brain Project [86], the Big Data for Neurosciences project [84], and BrainTrawler [25].

1.1 Problem Statement

Biologists create large collections of volumetric images of brains from different species, with specific genes visually expressed using modern labeling and staining techniques such as the GAL4/UAS or the LexA system in combination with fluorescence confocal microscopy. To ensure spatial alignment of the data, the sample images are registered onto a reference frame, a template brain.

Biologists need to find images where genes in user-specified spatial regions are strongly expressed in an interactive time frame and under different conditions. This can include the strength of the expression within the region of interest or nearby, considering the morphological variability of neurons, or the distance or overlap with segmented structures.

Cell sample collections with gene expression profiles are taken to determine dominant cell types and gene distributions in brain regions. All relevant metadata of images and cell samples are in most cases preserved in relational databases, which enable querying and filtering. However, given the spatial nature of the reference frame, traditional techniques for relational databases do not suffice for high resolution 3D spatial data and queries.

The storage, handling, and access of gene expression data in matrices of several gigabytes present a considerable challenge. Given the memory requirements, it is crucial to find and access only the necessary data. To determine the relevance of a given gene expression matrix, scientists rely on different factors like the spatial origin of the sample data or the accompanying metadata, including cell type, phenotype, and specimen sex. In conventional methodologies, accessing such information requires retrieving data from three separate sources: the spatial data source, the metadata source, and the gene matrix data source. The performance of data access is consequently hindered by this multifaceted approach. The connection and combination of grid data based on staining images and cell data representing regions in the grid open up new ways for knowledge discovery.

1.2 Requirements

The following requirements were specified due to the use cases defined in the application projects and the limitations of the existing indexing prototype:

General Requirements:

- Fast access to regions of interest in a large volume grid
- Capability to handle large collections of multivariate grid-based data
- Capability to handle large collections of region-based data
- Easy integration into various infrastructures
- On-the-fly data aggregation to minimize the memory requirements of queries

Mapping Requirements:

- Mapping of volume grid coordinates to data entries of index items
- Mapping of volume grid coordinates to regions
- Mapping of regions to multiple representations (region samples)
- Region samples must be independently accessible

Extensibility Requirements:

- Extensibility for different data types
- Extensibility for new data en- and decodings
- Extensibility for new data query types
- Adaptation of the indexing strategy to the data or new use cases

Data Storage Requirements:

- Support for different storage systems and environments
- Support for different data representations
  - Storage of grid-based data with a fixed payload size for each data entry
  - Storage of region-based data with multiple data entries (region samples)
- Encoding of different structural entities
- Encoding of dataset identifiers

1.3 Contribution

This thesis presents a novel implementation, including conceptual enhancements to the indexing prototype originally developed by Schulze et al. in 2012 [76], and builds upon the concept established by Bruckner [12] et al. and Solteszova et al. [79]. This concept involves using a *space-filling curve* to establish a locality-preserving order of the *voxel-based data* on the storage medium in combination with a page table for data lookup.

Schulze used two index data implementations: a novel one for a binary mask staining and one to store an object distance-field table, as also used by Bruckner and Solteszova. While the original prototypes demonstrated efficient data access, they had processing limitations due to a sequential, single-threaded read-process workflow that blocked the reading of subsequent data blocks during processing. Ganglberger et al. [26] used the same concept to store gradient vector flow data, with the goal of finding similar structures in the sample data collection where the structures are locally shifted.

The primary objective of this thesis is to generalize the concept used by Bruckner, Solteszova, Schulze, and Ganglberger and to provide a more versatile and extensible solution with an improved processing model. This includes enhancements to data handling capabilities, the encoding of structural types and dataset identifiers, and support for the same index data across multiple voxel grid coordinates (a *region*). This region data supports multiple variable-sized, complex, and independent data entries (*region samples*). The SPX framework efficiently handles multiple potentially overlapping regions with thousands of region samples.

To enable support for different data partitions or data entries for the same index entry at the same coordinates, the proposed concept introduces *data layers*. Layers allow multiple data entries for each index entry but in separate data areas, enabling conditional reading of the next data entry of the same indexed item at the same coordinate or region sample if required by the addressed query.

To facilitate the easy implementation of new index types and data representations, I defined standardized *codec* interfaces that hide the inner workings of the SPX framework from the extending user, and that allow for an easy and flexible integration of new data representations and queries. The two conceptually different data integrations, grid-based data and region-based data, are abstracted by the *region codec* and the *voxel codec* interfaces, respectively.

Taking advantage of the advent of multicore CPUs, parallel data processing has been incorporated into the SPX framework to enable parallel query data processing. Optionally, the additional processing power is used to reduce the index size and the amount of data to be read by applying state-of-the-art compression techniques.

A clear separation between data handling and data storage allows for the quick introduction of new data storage techniques. Combined with the space-filling curve interface,

which determines how the SPX framework orders the data, new indexing and storage optimizations depending on the type of index usage are possible.

The data entries for indexed items, as well as queries supporting the indexed data, depend on the index's purpose. Using the novel abstractions, I have implemented support for all before-mentioned index data types and queries on this data, including the binary mask staining, the object distance-field table, and the gradient vector flow representation. Build upon the novel region and data layer concepts, I created a *region codec* based implementation handling single-cell brain data samples representing brain regions. This data consists of two parts: the single-cell metadata, and the observed gene expression profile. The region codec's implementation allows to execute complex queries on this integrated data type.

1.4 Goals

The SPX framework is an essential core part of the Brain* framework [85]. It should enable the Larvalbrain project [86] to efficiently explore a large image collection of Drosophila melanogaster GAL4/UAS split line images using different data representations and query methods. These data representations include the binary image staining mask data used by Schulze et al. [76], the object distance-field table data used by Schulze and Bruckner and Solteszova et al. [12, 79], and the gradient vector flow representation introduced by Ganglberger et al. [26].

To give a short summary of the questions supported by the different data representations:

- *Which images have a strong staining inside the queried area?*
- *Which images have a similar staining compared to a reference image inside the queried area?*
- *Which anatomical segmented objects are in, near or overlap the queried area?*
- *Which images have anatomical and structural objects strongly expressed?*
- *Which images exhibit a comparable structural similarity within or close to the queried area?*

Explicit explanations of the research questions tackled by the different data representations can be found in the respective publications.

These use cases are all based on voxel grid data. Also part of the Brain* framework is BrainTrawler [25], for which SPX should provide the ability to answer the following questions - based on voxel grid and region data:

- *What is the cell type distribution in a query region?*
  Grid coordinates belong to brain regions, and each brain region exhibits a unique

composition of cell types as reflected in gene expression profiles. To determine the cell type within a particular brain region, one can make an educated guess based on the statistical gene expression distribution using tools such as HiReFIT [43]. Alternatively, the cell types are already available and saved in the samples' metadata.

- *What are the aggregated gene expressions in a region?*
  The proposed index structure should facilitate real-time aggregation of cell sample gene expressions inside user-defined regions. To calculate the mean/min/max expressions of a set of genes, one must examine all the cell samples collected from a particular brain region.

- *How to filter, categorize, and aggregate gene expression data according to metadata of single-cell samples?*
  Each cell sample possesses distinct meta properties, such as sex, phenotype, genotype, or dataset affiliation. One aim is to filter and categorize the potential results based on these properties or the expression strength of genes.

Query processing and communication between the SPX framework and the querying client should be in a data representation-independent uniform way, and the introduction of new data representations and queries should be easily possible in a standardized way.

1.5 Thesis Overview

This thesis begins by providing a background chapter that familiarizes the reader with the biological context in which the proposed spatial indexing framework is applied. The subsequent chapter delves into related work in the field, offering an overview of existing approaches and their limitations. This is followed by an overview of key terms essential to understand the following chapters. After that, an overview of the framework is provided, followed by a methodology chapter, which outlines the development methodologies and describes the methods employed in the development of the specific spatial indexing approach. This is succeeded by the implementation details of the approach. In the subsequent chapter, I present two applications, BrainBaseWeb [86] and BrainTrawler [25, 27], that currently employ the SPX framework, and the various index types and queries that are in use. Finally, the thesis concludes with an evaluation of the spatial indexing approach, examining its effectiveness, limitations, and potential future directions for improvement and development.

CHAPTER 2

Background & Context

The brain remains a largely unexplored frontier, and neuroscience as a research discipline is still in its early stages. The quest to understand the brain and its functionality has led to numerous discoveries related to cognitive and behavioral conditions, as well as the connections between various illnesses of the brain. Neuroscience aims to shed light on the enigmatic workings of the brain, including the intricate relationship between firing neurons and resulting behavior, and the functional rationale behind the network connectivity of different brain regions. This pursuit relies on observing chemical actions and reactions in neuronal circuits, facilitated by advanced genetic techniques and state-of-the-art imaging procedures, as well as studying gene distributions in these regions. This chapter provides a brief overview of the biological background to establish a foundation for understanding the use cases of the proposed spatial indexing framework.

2.1 Brain Biology

The brain is a vital component of the central nervous system, comprising structural connectivity tissue and functional tissue, consisting of distinct cell types.

2.1.1 Cell Types

According to Purves et al. [69], neurons and glial cells are the two main cell types in the brain. These cells perform essential functions in terms of anatomy, electrophysiology, and molecular aspects. Examining the fundamental operations performed by neurons and glial cells can yield insights into a multitude of inquiries regarding brain function.

Neurons and glial cells form intricate neural circuits that constitute the fundamental building blocks of neural systems. These systems serve three primary functions: processing of sensory inputs, controlling motor functions, and linking the sensory and motor systems

through associational circuits, which is also responsible for higher-order cognitive functions such as perception, attention, rational thinking, emotions, and other critical processes.

Neurons

Neurons are the primary functional units of the nervous system that transmit and process information through electrical and chemical signaling. They possess unique features such as excitability, conductivity, and plasticity, which enable them to encode and store information by modifying their structure in response to experience and learning. This ability allows the brain to adapt to new situations and form new memories.

The structure of a neuron, as described by Purves et al. [69], consists of a cell body or soma, dendrites, and an axon. Dendrites receive information from other neurons, while the axon transmits information to other neurons or effector cells, such as muscles or glands. The axon is insulated by a myelin sheath, which facilitates the rapid transmission of signals. Bundles of axons are referred to as fascicles. The transmission of signals between neurons occurs at specialized structures called synapses, where neurotransmitters are released from the presynaptic neuron and bind to receptors on the postsynaptic neuron, crossing the synaptic cleft in the process.

There are several types of neurons, including sensory neurons that transmit information from sensory receptors to the brain, motor neurons that control muscle movement, and interneurons that facilitate communication between neurons. Mirror neurons are a special type of neuron thought to play a role in empathy, social learning, and imitation.

Neurobiologists also study different developmental stages of animals, e.g., the younger L1 stage of the *Drosophila melanogaster*, which develops into the L3 stage. Particularly in the younger stages, neurons are not fully developed yet. The immature or developing nerve cell that eventually differentiates into a mature neuron is called a neuroblast [29]. These cells are derived from stem cells and have the potential to develop into any type of neuron found in the nervous system.

Synapses

Synapses are structures that facilitate the transmission of signals between neurons and other cells by releasing neurotransmitters from the presynaptic neuron. The neurotransmitters bind to receptors on the postsynaptic neuron, allowing the signal to be propagated across the synaptic cleft. This process is fundamental to the functioning of neural circuits.

Synapses also play a vital role in enabling learning and memory formation by strengthening or weakening connections based on experience. In addition to neurons, glial cells such as astrocytes are actively involved in modulating synaptic activity. Synapses are densely concentrated within neuropils, where they form intricate networks that are essential for integrating sensory input and coordinating motor output. The complex branching of

neuronal dendrites and axons, known as arborizations, further facilitates the formation of synaptic connections.

Glia Cells

Glia cells are a diverse group of non-neuronal cells that support and protect neurons in the nervous system. They are involved in many important functions, including regulation of the extracellular environment, myelin formation, neuronal guidance during development, synaptic plasticity, and immune defense against pathogens and injury.

According to Purves et al. [69], glial cells can be categorized into three main types. Astrocytes, as described by Molofsky et al. [57], support synaptic transmissions, play a role in the blood-brain barrier, and participate in tissue repair and scarring. Oligodendrocytes surround axons, isolating them from the surrounding environment and speeding up electrical signaling. Microglia, as described by Paolicelli and Gross [64], are the primary immune defense in the nervous system and are involved in synaptic pruning during development and in response to injury.

Neuropils

One semantic layer above the two basic cell types in the hierarchical organization of the brain are the neuropils. Neuropils are complex networks that function as integrative systems for the anterior sense organs. They consist of densely interwoven nerve fibers, dendrites, axons, and their branches, as well as synapses, filling the space between glial cells and neuronal cell bodies [69]. They serve as the primary sites for the formation and maintenance of synaptic connections between neurons.

Arborizations

Neuronal circuits can be identified by overlaps of arborizations, which are the intricate branching structures of a neuron's dendrites and/or axons. These branching structures facilitate the formation of complex networks of connections between neurons and are critical for neural communication and information processing in the brain [69, 78]. Dendritic arborizations of neurons receive input from other neurons, while axonal arborizations transmit signals to other neurons.

All these structures hold great significance in neurobiological research. Creating images where one can depict these structures and understand their connectivity can provide valuable insights into the brain and its functional organization.

2.1.2 Data Types

Neurobiological research generates a wide range of data types; therefor, I only provide a brief description of the data types used in this thesis.

2.1.2.1 Single-Cell RNA Data

Single-cell RNA sequencing (scRNA-seq [27]) is a technique that enables the analysis of gene expression at the level of individual cells. This method involves sequencing the RNA of single cells extracted from biopsy sites distributed throughout the brain. The gene expression profiles obtained from these samples allow for the categorization of cells into distinct cell types. However, this categorization process can be complex, necessitating the use of tools such as HiReFIT [43] or SINCERA [30] to determine the likeliest cell type (e.g., astrocyte, oligodendrocyte, glia, neuron, etc.) for a given sample. The aim of this analysis is to identify the predominant cell types in a specific brain region, and to elucidate the connections between gene expression, brain regions, and neurological functions. It should be noted that the cell samples collected during scRNA-seq do not have assigned coordinates, but rather represent the brain region from which they were extracted. The SPX framework was used for scRNA-seq data of mouse and human.

2.1.2.2 GAL4/UAS Staining Images

Biologists generate extensive sets of volumetric images using the GAL4/UAS labeling technique, which highlights specific cell types under confocal microscopy. These images are compiled by stacking individual image slices together and the meta-data is stored in a database. To ensure spatial comparability, the images are registered onto a common registration template image [60, 71], which is typically an average of many manually aligned images. An example rendering of a GAL4/UAS staining image including the registration template can be seen in Figure 2.1.

GAL4/UAS Staining

Introduced by Brand and Perrimon in 1993 [10], the GAL4/UAS system is a genetic tool used in model organisms such as *Drosophila melanogaster* to manipulate gene expression in specific tissues or cells. It utilizes two genetic constructs: the GAL4 driver and the UAS responder.

According to Duffy [20], the GAL4 driver protein is a transgenic construct containing a promoter sequence that drives the gene expression of the yeast transcription factor GAL4. The GAL4 protein binds to specific DNA sequences, activating the expression of downstream genes. The promoter of the GAL4 driver can be designed to be specific to particular tissues or cell types, enabling selective gene expression in those cells. If the GAL4 driver is present in a cell or tissue, it activates the Upstream Activator Sequence (UAS) responder.

These GAL4-expressing neurons can be visualized using a fluorescent marker like Green Fluorescent Protein (GFP), as described by Chalfie et al. [14]. As a result, only neurons that express GAL4 will also express GFP, enabling researchers to visualize and study these specific neurons and their connections under a confocal microscope. This technique is commonly referred to as staining.

Figure 2.1: GAL4/UAS staining data of a Drosophila melanogaster in L3 stage rendered with Direct Volume Rendering (DVR). - *Image rendered in www.larvalbrain.org*

This system allows researchers to study the function of genes in a tissue- or cell-specific manner, providing insights into the role of genes in development, behavior, and disease. The GAL4/UAS system has been widely used in Drosophila research and has also been adapted for use in other organisms, including mice and zebrafish.

GAL4 Split Lines

As explained by Luan et al. [48], GAL4 split lines are a refinement of the GAL4 system that allows more precise targeting of gene expression in cells. The GAL4 transcription factor is split into two parts: the DNA-binding domain and the activation domain. Each part is expressed under the control of different tissue-specific promoters, resulting in two distinct subsets of cells expressing each part. If both fragments are co-expressed in the same cells or tissues, they reconstitute the functional GAL4 protein and activate downstream gene expression. This system provides more precise control of gene expression in specific cell populations by targeting expression to cells that express both fragments of the split-GAL4 protein, rather than just those expressing the GAL4 driver construct.

LexA Staining

LexA staining operates on a similar principle as the GAL4 system, utilizing the LexA yeast transcription factor instead of GAL4. LexA binds to specific DNA sequences known as LexA operators (LexO), which are equivalent to the UAS in the GAL4 system but distinct in their binding sequences. This distinction allows for targeting different cell types. The LexA system also involves two components: a driver line expressing the LexA transcription factor in specific cells or tissues, and an effector line carrying a LexA operator upstream of a gene of interest.

When the driver and effector lines are crossed, the LexA transcription factor binds to the LexA operator, activating the gene of interest's expression specifically in the target cells or tissues. Like the GAL4 system, the LexA system is versatile, being used for labeling specific cells or tissues, manipulating gene expression, and studying neural circuits.

Ting et al. [82] and Yagi et al. [92] used the LexA system to complement the GAL4 system, providing a second independent transactivator function to the GAL4/UAS system. This allows for intersectional labeling, enabling even finer control over expression patterns.

2.1.3 Confocal Light Microscopy

High-resolution laser microscopes are indispensable tools for capturing detailed images of microscopic brain structures in model organisms such as *Drosophila melanogaster*. Laser scanning microscopy, often used in conjunction with fluorescence markers like Green Fluorescent Protein (GFP), is particularly effective for visualizing specific biological structures. In this process, the laser beam of the microscope is focused by the objective lens and directed into the tissue, where it excites the fluorescent molecules.

The fluorescence emitted in response, which exhibits a longer wavelength than the excitation light, is separated from the excitation light by a dichroic mirror. This mirror is designed to reflect the excitation light and transmit the longer-wavelength fluorescence.

Although the fluorescent tissue emits light in all directions, the strongest signal originates from the focal point, while the surrounding tissue contributes a weaker background signal. This background signal can be effectively eliminated by a pinhole, thereby increasing the clarity of the final image.

The ultimate resolution of the image depends on the specific microscope used. A volumetric image is generated by stacking multiple two-dimensional microscopy images, or slices, on top of each other. During image acquisition, it is critical to prevent any deformation of the specimen under the microscope. For further details, readers are referred to the *Handbook of Biological Confocal Microscopy* [65].

2.1.4 Neuroscientific Atlases

Neuroscientific atlases are comprehensive databases or maps that provide intricate information about the anatomical structure, connectivity, and function of the nervous system. They serve as reference points for researchers and clinicians, aiding in the identification of different brain structures and the understanding of their functions. These atlases are constructed using a variety of techniques such as histology, microscopy, magnetic resonance imaging (MRI), and positron emission tomography (PET).

Histological atlases leverage tissue staining and microscopy to offer detailed insights into the cellular and molecular composition of the brain and nervous system. These atlases are instrumental in identifying diverse types of cells, their specific locations, and their interconnections. They prove particularly beneficial for investigating the structure and function of the brain at a cellular level. While some atlases concentrate on specific regions or functionalities of the brain, others provide a comprehensive overview of the brain in its entirety.

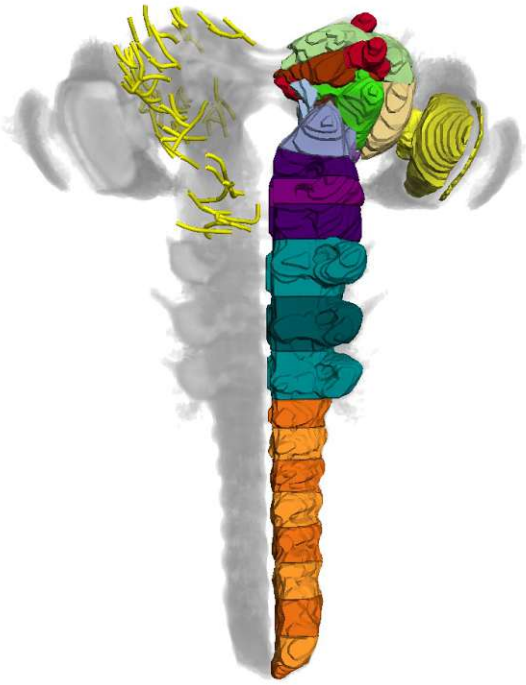

Figure 2.2: A high-level neuropil atlas of the Drosophila melanogaster in the L3 larvae stage with a few axon tracts shown on the left side. - *Atlas rendered in www.larvalbrain.org*

Digital interactive atlases made possible by advancements in imaging technology, visualization techniques, and data analysis allow for interactive exploration of the brain and can incorporate multiple imaging modalities for a more detailed view. They can

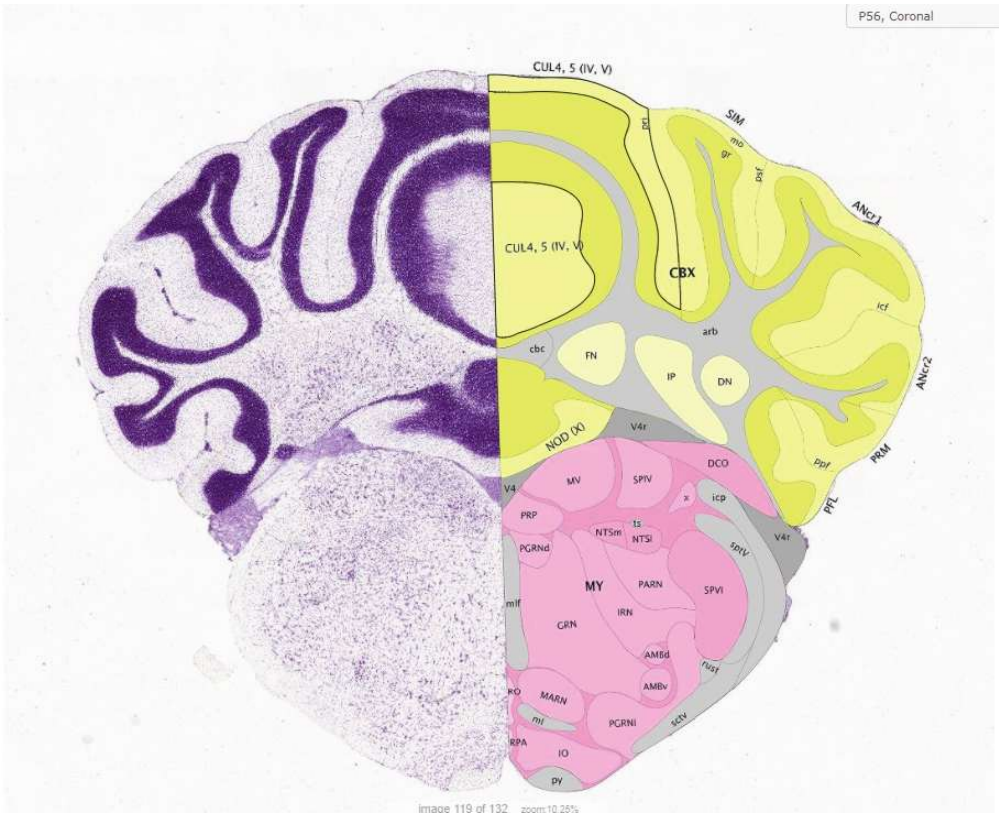

Figure 2.3: Allen Mouse Brain Atlas - P56, Coronal - Lobules IV-V, molecular layer - *View taken from atlas.brainmap.org*

integrate data from various sources, such as gene expression profiles or connectome data, providing a more holistic understanding of brain structure and function. An example for the Drosophila melanogaster in the L3 larvae stage using polygonal meshes for neuropils can be seen in Figure 2.2. Another example view taken from the Allen Mouse Brain Atlas can be seen in Figure 2.3.

Examples for brain atlases include:

- **Human**
  Big Brain Atlas [4], Allen Human Brain Atlas [39]
- **Mouse**
  Allen Mouse Brain Atlas [40],
  Genome-wide atlas of gene expression in the adult mouse brain [47]
- **Drosophila**
  FlyBrain [5], Janelia FlyEM atlas [13], Virtual Fly Brain [63], Larvalbrain [86]

2.2 Image Handling

This section provides an overview of the technical challenges in the field of image handling, particularly in neurobiological contexts.

2.2.1 Intensity Normalization

Intensity normalization is a postprocessing step often applied at the dataset level for a collection of volumetric images. Its purpose is to reduce variability in image intensity values that are not related to the structures being imaged. This variability can arise from various factors, including differences in imaging equipment, imaging conditions, or individual differences in tissue properties. By normalizing the intensity values, these sources of variability are minimized, facilitating the comparison and integration of data from different sources.

2.2.2 Template Volumes

In neuroimaging, a template refers to a standard or reference volume used for the alignment of confocal microscope image stacks from different sources. The template represents a *standard* or *average* brain, constructed by averaging several hand-picked, well-aligned images. This is crucial because individual brains can vary significantly in size, shape, and natural variability, as well as experience distortions during the image extraction and placement process for imaging.

The registration template also defines the data grid for the image data, specified by the template's dimensions and the number of voxels in each dimension. This grid is used to calculate the transformation between the template and the image data. The transformation is then applied to the image data to align it with the template, a process known as registration.

2.2.3 Volumetric Image Registration

Modern high-resolution imaging methodologies facilitate the creation of detailed volumetric images by generating a series of 2D slice images at consistent intervals. This image stacks are then compiled into a 3D volume. In neurobiological studies, specimens such as *Drosophila melanogaster* brains or larvae are exceedingly small and challenging to orient and align accurately prior to slicing. For larger brains, such as the human brain, the slicing process involves mechanical operations on a frozen brain, which may result in distortions like tears and pulls.

Image registration, or the alignment of 3D volumes onto a reference template, is a crucial step in integrating data from different imaging modalities or conducting population-based studies with multiple subjects. The process involves spatially transforming (or warping) the images to preserve the correspondence between their anatomical or structural features. The objective is to transpose various data sets into a common coordinate system for accurate comparison and integration.

Registration can be broadly classified into two categories: rigid and non-rigid. Rigid registration, permitting only translation, rotation, and scaling, is generally applied when dealing with rigid objects, such as bones. Non-rigid registration, in contrast, allows for local deformations and is typically used for soft tissues like the brain, which can change shape between different imaging sessions or among different individuals.

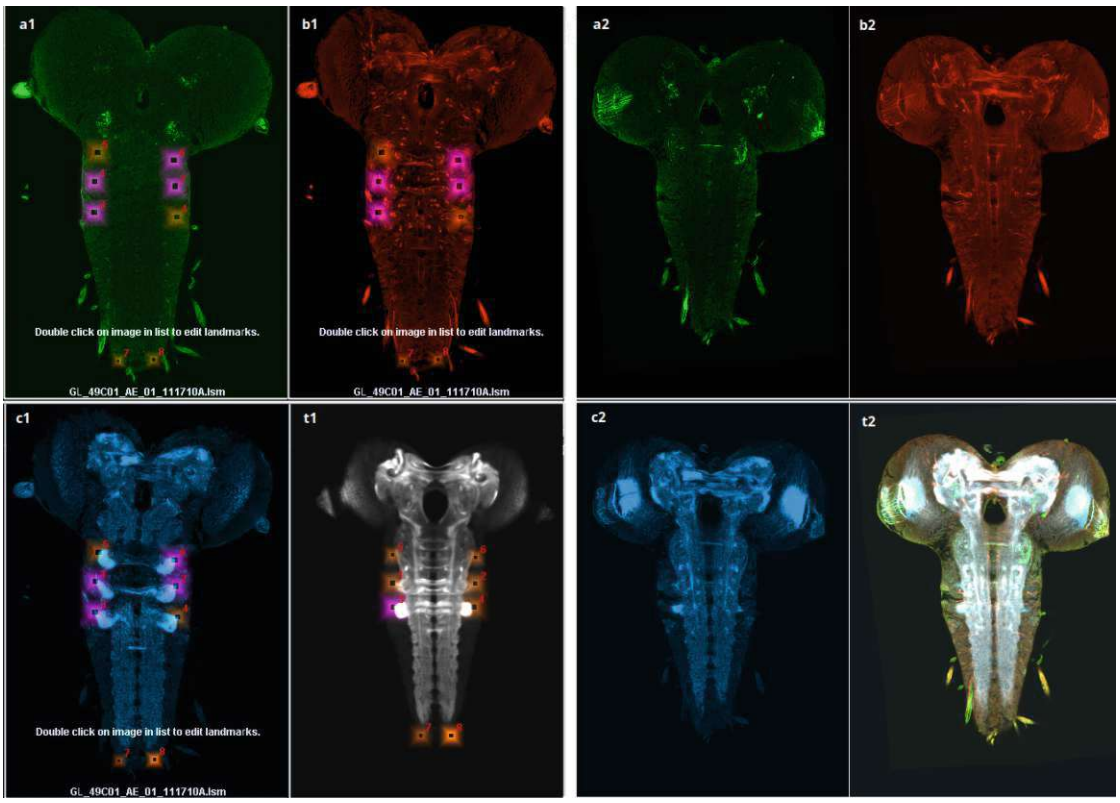

Figure 2.4: Landmark based registration using three channels.

Channel images before the registration with the landmarks set *(1)*, and after the registration *(2)*:
a) anti-GFP
b) antibody
c) neuropil
t1) template with reference landmarks
t2) combined control view with the template
- *Images rendered in BrainWarp, the registration management tool of the Brain* framework [85].*

The registration process typically comprises three steps:

1. **Preprocessing** the images to eliminate noise, artifacts, or other distortions that could impede the registration process.
2. Identifying corresponding features (**landmarks**) in the images to estimate the transformation.
3. Estimating the **transformation parameters**, which could involve rigid, affine, or deformable transformations depending on the degree of deformation between the images.

In neurobiological research, it is common practice to register obtained image data onto a reference template to align images of brain structures or neurons across different subjects or experimental conditions. This alignment enables quantitative analysis and comparison of the data. A combination of rigid and non-rigid image transformation is often required to achieve the best registration results. Specialists utilize optimization methods to automatically register larger collections of images using tools such as Elastix [44] with optimized parameters, as demonstrated in Larvalign [60]. However, manual landmarks are often necessary to achieve a usable result.

2.2.4 Image Segmentation

Segmentation is the process of partitioning an image into multiple segments, where each segment corresponds to a different object or region of interest. In neuroimaging, segmentation is often used to identify and isolate specific brain structures, such as different arborizations, neuropils, or even individual neurons. This is an essential step in many neuroimaging analyses, as it allows for the measurement and comparison of specific brain structures across different samples. Furthermore, it forms the foundation of the aforementioned neurological atlases.

2.2.5 Visualization

Various methods are available for representing neurological data, with the most frequently used approach involving the rendering of slices, often supplemented by volume rendering. In both cases, rendering the expression image is done with a template image in the background to provide biological context. A *transfer function* is applied to map the expression values to colors and opacities, enabling a focus on certain expression patterns and strengths. By overlaying multiple images, additional details and insights can be obtained for the desired structures.

A significant amount of research has been conducted in the field of medical and biological volume rendering, and a detailed discussion of these methods is beyond the scope of this thesis. Standard approaches include Direct Volume Rendering (DVR) [21] as used in Figure 2.5, Maximum Intensity Projection [21, 59], First Hit Rendering [21], and Maximum Intensity Difference Accumulation [11].

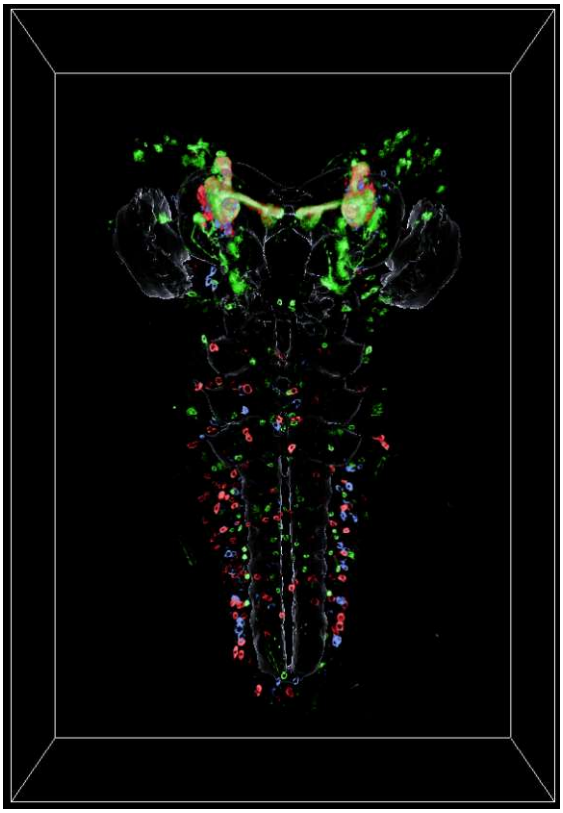

Figure 2.5: Direct volume rendering of three sample images of the anti-GFP channel of the Drosophila melanogaster - L3 stage. Template rendered as hologram mesh. - *Image rendered in www.larvalbrain.org*

Summary

The integration of different types of neurobiological data, from individual neurons to whole brain structures, is a complex task that requires careful preprocessing, alignment, and integration of data from various sources. Techniques such as the GAL4/UAS and LexA systems enable the specific labeling of different cell types, while imaging techniques like confocal microscopy provide high-resolution images of these labeled cells. Image registration techniques are used to align these images with a common template, facilitating accurate comparison and integration of data from different sources. Finally, computational tools are employed to segment and classify the data, identifying different brain structures and cell types based on their unique patterns of gene expression. Together, these techniques form a powerful toolkit for understanding the complex organization and function of the brain at multiple scales, from individual neurons to whole brain networks.

CHAPTER 3

Related Work

This chapter explains spatial indexing, discusses current state-of-the-art techniques, and presents various approaches used. It also covers the application of spatial indexing in neurological research and provides a brief overview of the tools and resources available in this field.

3.1 Spatial Indexing

Zhang and Du [94] state that spatial indices are data structures which allow the efficient storage and retrieval of spatial data and objects. Without indices, data access would require a sequential scan, reading all available records sequentially until the requested data is found. It involves dividing the indexed space into smaller, discrete regions. Spatial objects are assigned to one or more regions based on their location, or each location in the indexed space gets one or multiple values assigned. This allows for faster queries and operations on large datasets, as only the regions containing relevant objects and values need to be accessed. Common spatial indexing techniques include grid indexing, quad- or octree indexing, and R-tree indexing. These techniques are used in various applications, such as geographic information systems, computer graphics, and database management systems. Figure 3.1 illustrates the speed difference between spatial indexed and non-spatial indexed queries in a GIS database.

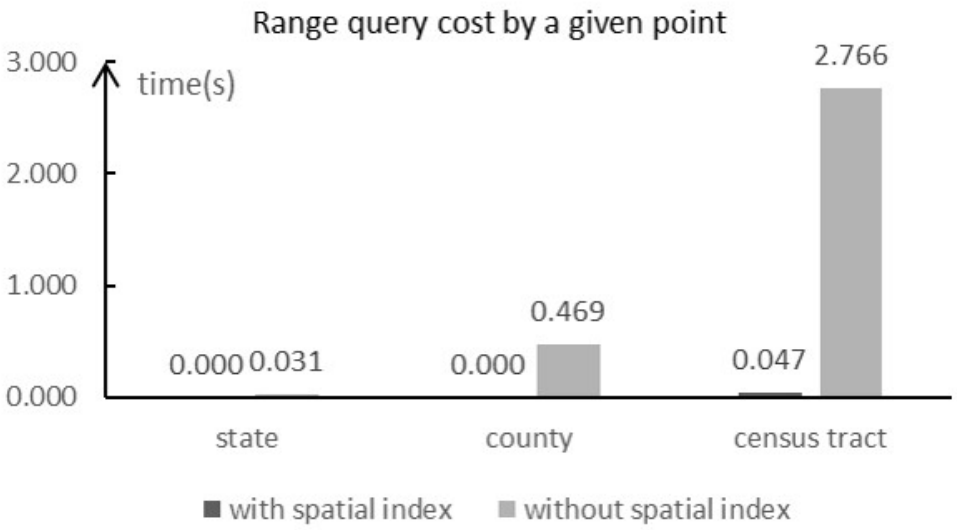

Figure 3.1: Point query - spatial indexed vs. sequential scanning of GIS data in MySQL using 2017 TIGER national geodataset. - *Image from [94].*

Zhang and Du [94], as well as Güting et al. [31], list the following three basic spatial query types:

- **Range Queries** search for objects within a specific geometric shape or range, such as finding all the data entries that fall within a certain distance of a given point.
- **Nearest Neighbor Queries** search for the closest data entries to a specific point of interest, such as identifying the closest hospital or gas station to a user's current location.
- **Spatial Joins Queries** search for objects that spatially interact with each other, with predicates like intersection, containment, or adjacency. They combine two sets of data by matching their spatial relationships, such as finding all the customers who live within a certain distance of a store location.

Objects typically possess intricate shapes and are frequently approximated during search traversal. The prevalent approximation method is the minimum bounding rectangle, or the minimum bounding box in 3D, characterized by the coordinates $x_{min}, y_{min}, (z_{min})$ and $x_{max}, y_{max}, (z_{max})$.

Following the basic categorization of spatial index structures proposed by Rigaux et al. [70], one can distinguish the following types:

- **Data-driven structures** partition the embedding space into groups of spatial objects using their minimal bounding rectangles/minimal bounding boxes.
- **Space-driven structures** partition the space into cells with query regions defined by the overlap and intersection of cells.

3.1.1 Data-Driven Indexing Structures

Following Rigaux et al. [70], Zhang and Du [94] state that in data-driven spatial indexing structures the relationship of object containment structures like the minimal bounding rectangle/box. The structures are build up while inserting the data, and are designed to provide efficient and effective query processing by adapting the indexing to the specific properties of the data (e.g., density, distribution, and dimensionality). Examples of data-driven spatial indexing structures include the BSP-tree and the R-tree.

BSP-Trees

The Binary Space Partitioning Tree (BSP-Tree) [1, 61] recursively divides the space into two halves using a hyperplane, which is chosen in such a way that it minimizes object intersections. The hyperplanes can be oriented in any direction, including being axis-aligned. Each node in the tree represents a sub-region of the space, and each sub-region is recursively partitioned into two parts until a termination condition is met. One example is the kd-tree.

The k-dimensional (kd) Tree recursively partitions the data into smaller regions, as seen in the example in Figure 3.2. Each node of the tree represents a hyperplane subdividing the parent into two subspaces. The placement of the splitting plane is chosen based on the data's distribution. The plane is always orthogonal to the parent's hyperplane and aligned with one of the axes.

Other types of BSP-trees specialize for different spatial indexing problems including region trees specifically designed to optimize range searching problems, interval trees or segment trees for line intersection problems. Güting lists the alternatives in *An Introduction to Spatial Database Systems* [31].

R-Tree

The R-tree [32] is the most popular solution for indexing objects. Its hierarchy is organized by nested minimum bounding rectangles (MBRs) in 2D space, or by minimal bounding boxes (MBBs) in the volumetric space of objects. The parent node always contains all the child MBRs/MBBs, and the root node of the tree contains an MBR/MBB that encompasses all of the data points.

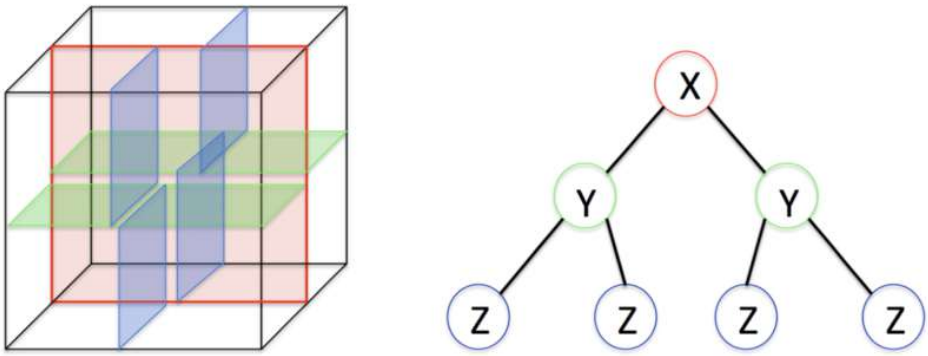

Figure 3.2: Cube to kd-tree. - *Image from [15].*

The R-tree is a balanced search tree, meaning that all leaves are on the same level. It is a challenge to build balanced R-trees in which the MBRs/MBBs are as little as possible overlapping and the boxes do not cover much empty space. Each node is inserted in the MBR/MBB that grows the least. If the number of nodes in an MBR/MBB exceeds a certain limit, the MBR/MBB is split into two parent nodes.

The R-tree is not efficient for point data, as the MBRs/MBBs can be very large and cover a lot of empty space. In this case, the R-tree can be replaced with a kd-tree, which is more efficient for point data. However, the R-tree is more efficient for data with higher dimensionality, such as polygons.

Several variants of the R-tree exist, each with different optimizations to cover a wide range of use cases. Examples include the X-tree [9], which uses a more sophisticated splitting technique, allowing it to handle high-dimensional data more efficiently. Another extension is the R*-tree [8], which has higher build-up costs but improved query performance by reducing overlap and coverage of bounding boxes through more sophisticated insertion and splitting strategies, resulting in a more evenly balanced tree structure. Other examples include the R+-tree and the Hilbert R-tree.

While space partitioning trees are a thriving area of research with the potential to introduce innovative approaches to spatial indexing, the focus of this thesis is primarily to handle point data and to enhance and expand the existing grid-based prototype and concepts.

3.1.2 Space-Driven Indexing Structures

Space-driven spatial indexing data structures are based on the partitioning of space into a set of non-overlapping cells. The data is then assigned to these cells based on their location in space. This allows queries to be defined by specifying a region of interest as a set of cells that overlap with the query region. Generally, they consist of tuples

of $< \vec{c}, o >$ where $\vec{c}$ is a coordinate vector and o is an object having an entry at this coordinate.

According to Zhang and Du [94], space-driven data structures are often used for range queries, where the goal is to retrieve all data objects that fall within a specified region of space. Examples of space-driven data structures include grid-based indexing, quadtree-based indexing, and octree-based indexing.

The main advantage of space-driven indexing structures is their ability to quickly eliminate large portions of the search space, as it is easy to determine which cells do not intersect with the query region. However, they may have worse performance if the data is not uniformly distributed in space, since some cells may contain much more data than others.

Spatial Grids

Samet [74] states that a spatial grid index partitions the space into cells. There are several types of grid-based indices, including regular grids, hierarchical grids, and adaptive grids. Regular grids, like in Figure 3.3, have fixed cell sizes and are simple to implement, but may not be efficient for datasets with varying spatial densities. Hierarchical grids, on the other hand, use nested grids with varying cell sizes to achieve a more efficient search process. Adaptive grids dynamically adjust the cell size based on the spatial density of the data points, allowing for more precise and efficient queries. A benefit of grid-based indices is their ability to easily determine which cells intersect with the query region.

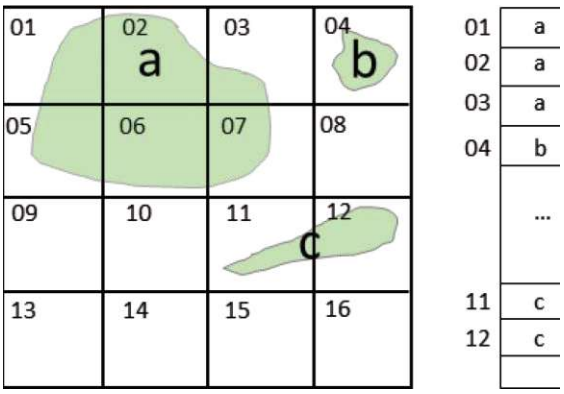

Figure 3.3: Spatial indexing using a regular grid mapping.
- Image from [94].

Hierarchical Grid Indices

Hierarchical Grid Indices, or multilevel grid indices, partition the space into a hierarchy of increasingly finer grids, with each level containing a different number of cells depending on the amount of data inside.

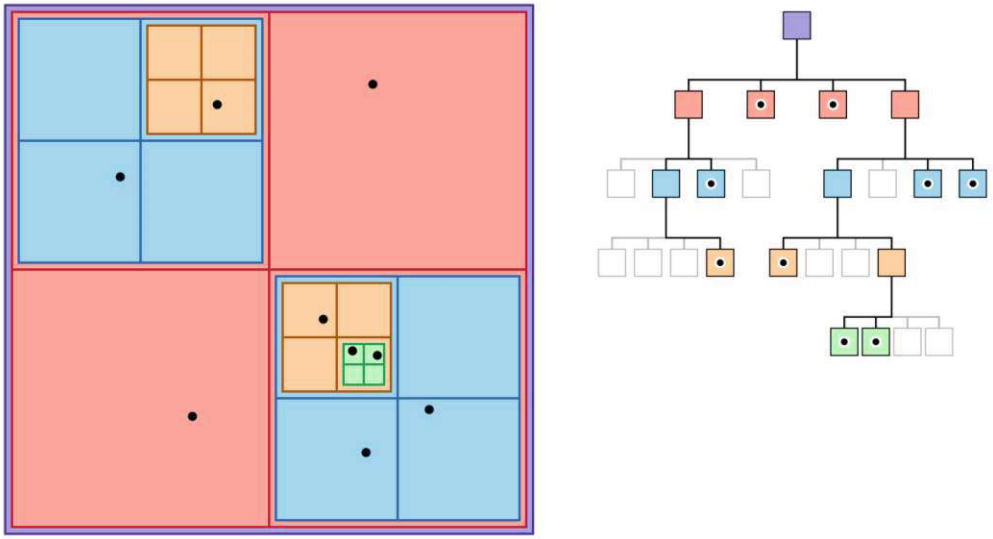

Figure 3.4: Square to quadtree. - *Image from [19].*

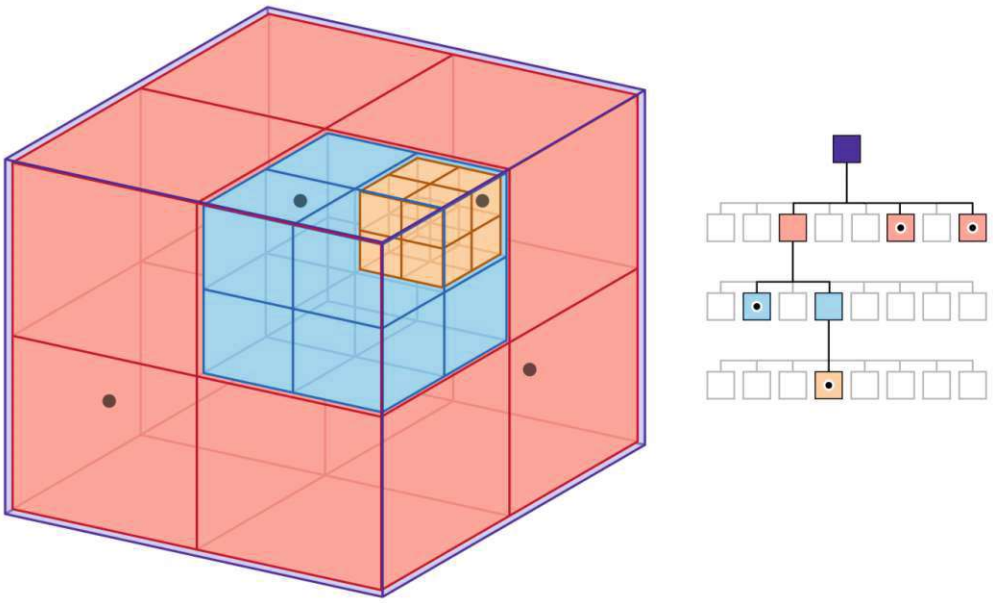

Figure 3.5: Cube to octree. - *Image from [18].*
I adjusted cube 6's position in the first row of the tree from the 5th place to the 6th.

Quadtrees/Octrees are the most common type of hierarchical grid indices. A quadtree is a two-dimensional grid that recursively subdivides each cell into four equal smaller cells. The same concept can be applied to three dimensions to create an octree. They represent a flexible grid with refinements and variable resolution in areas with higher object density, making them particularly useful for applications where the density of data points varies across the space. The splits are always in the center of a cell. One node is

either a leaf or a parent to four (quadtree) or eight (octree) child nodes. Examples for a quad- and an octree can be seen in Figures 3.4 and 3.5.

One downside of hierarchical grid indices is that they can become inefficient for large datasets, as the number of cells can grow exponentially with the depth of the hierarchy. To address this issue, various techniques have been proposed, including hybrid indices that combine hierarchical grids with other index structures. One example is the approach by Herzbergers et al. [35], where they use an octree to quickly determine which fixed data grids ("bricks") need to be streamed from the server and at what resolution.

Fixed Grid Indexing

Fixed grid indices partition the spatial domain into a grid prior to data insertion. This grid can be regular or irregular, and each grid cell represents a spatial region and is assigned a unique identifier. Data points are then assigned to the corresponding grid cells based on their spatial locations. This approach allows for searching only the relevant grid cells without any tree traversals or intersection tests. An example of a fixed grid index is the geographic coordinate system, where landmarks, streets, or houses are assigned specific coordinates within a fixed grid. Extending the geographic coordinate system to a three-dimensional space is done by using digital elevation models which assign a height value to each coordinate. Fixed grid indices are especially useful in applications where the spatial space is known in advance and does not change.

For efficient query management within a fixed grid, it is critical to arrange the data in a manner that respects spatial locality. This means that areas likely to be queried together should be grouped and stored in adjacent data blocks to optimize the use of cache hierarchy and read-ahead caching. This can be achieved through the use of a space-filling curve, as done by Solteszova et al. [79], Bruckner et al. [12], Schulze et al. [76], and Ganglberger et al. [26]. The primary data source used by Schulze et al. [76] and Ganglberger et al. [26] inherently defined a fixed grid due to the resolution of the indexed GAL4/UAS staining slice stacks. As the data is noisy and lacks structure, applying other data-driven concepts is unlikely to be feasible in this case.

3.1.3 Space-Filling Curves

Space-filling curves, invented by Giuseppe Peano in 1890 [66], are a class of mathematical functions used to map a higher-dimensional space onto a one-dimensional space in a way that preserves the spatial proximity of the points in the higher-dimensional space. In a data grid, this means that two adjacent cells have a high probability of being saved close together after the mapping, making fixed grid indexing an ideal application for space-filling curves. Additionally, it is possible to use efficient search mechanisms based on the linear order in multiple dimensions by applying them to the mapped curve. Some well-known examples of space-filling curves include the Hilbert curve, Peano curve, Moore curve, Sierpinski curve, and Morton curve, each with its unique properties and applications.

The Hilbert Curve

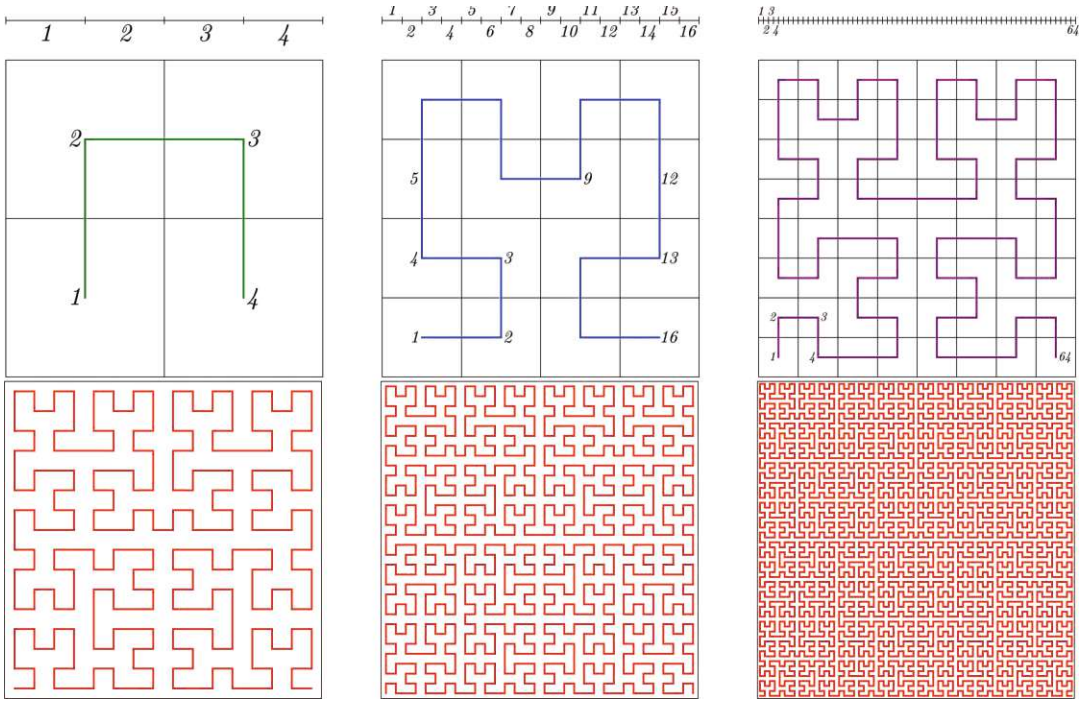

Figure 3.6: The first six iterations of the 2D Hilbert curve. - *Image by user Braindrain000 for https://commons.wikimedia.org/.*

The Hilbert curve, introduced by mathematician David Hilbert in 1891 [36], is a well-known space-filling curve. It is a continuous curve that passes through every point in a two-dimensional square, or three-dimensional volume, while preserving spatial proximity. This means that points close to each other in the original two-dimensional or three-dimensional space often remain close in the one-dimensional space after mapping. Figures 3.6 and 3.7 visualize several iterations of the Hilbert curve in two and three-dimensional space.

While there is a clearly defined Hilbert curve for two-dimensional space, Hilbert himself did not extend this concept to multiple dimensions. As pointed out by Haverkort [33, 34] and Alber and Niedermeier [3], the task of generalizing the Hilbert curve to multi-dimensional spaces can result in a multitude of solutions. Various solutions are proposed by researchers like Kamata et al. [41, 42], and Sagan [73]. These solutions originally revolve around scanning a cuboid region with a side length that is a power of two, but this approach involves computationally expensive recursive algorithms.

Alber and Niedermeier [3] present an extension to Hilbert's work that applies to arbitrary dimensions. Similarly, Zhang and Kamata [93] propose non-recursive solutions that utilize Gray code ordering and iterative refinement. Despite its usefulness, there is no direct formula to compute the Hilbert curve. Consequently, one must compute the entire

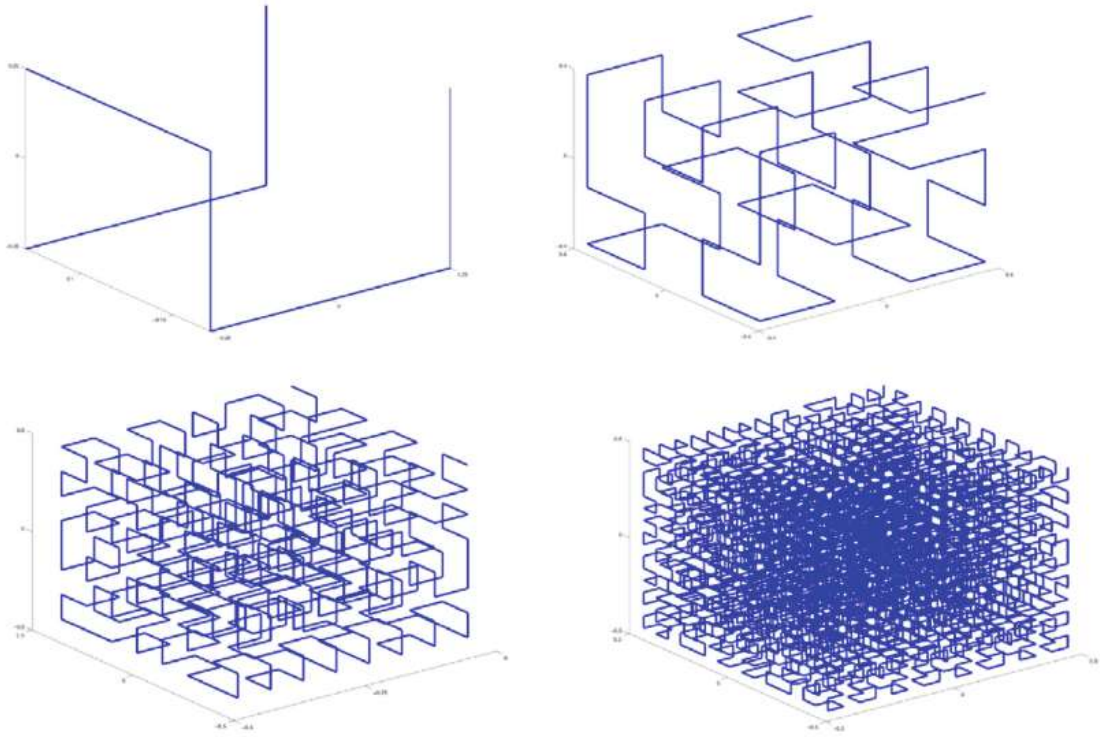

Figure 3.7: The first four iterations of the 3D Hilbert curve. - *Image from [53].*

curve for the cuboid in all scenarios. The calculation of the Hilbert curve and its reverse mapping in 3D space is both computationally intensive and complex. In this regard, the Morton curve, also known as the Z-order curve, is a more efficient alternative, being easier and faster to compute.

The Z-Order Curve

The Z-order curve, also known as Morton curve, was developed by Guy Morton in 1966 [58]. This curve follows a Z-shaped pattern, exemplified in Figure 3.9, and has found widespread applications in fields such as computer graphics, databases, and scientific computing.

The Z-order curve functions by interleaving the bits of the coordinates of each point, resulting in a singular binary number, as illustrated in Figure 3.8. This number is subsequently used to determine the position of the point along the curve. The order it generates is equivalent to a depth-first traversal of a quadtree or octree, and is therefore well-defined in volumetric space as well. Both the Morton curve and quadtree/octree methods recursively partition the space into four or eight quadrants, with each cell assigned a unique coordinate. Compared to the Hilbert curve, the Z-order curve offers the advantages of easier computation and a more straightforward inverse mapping process. However, it has a slight drawback in that it has less efficient locality-preserving properties,

which could potentially lead to less efficient use of cache memory.

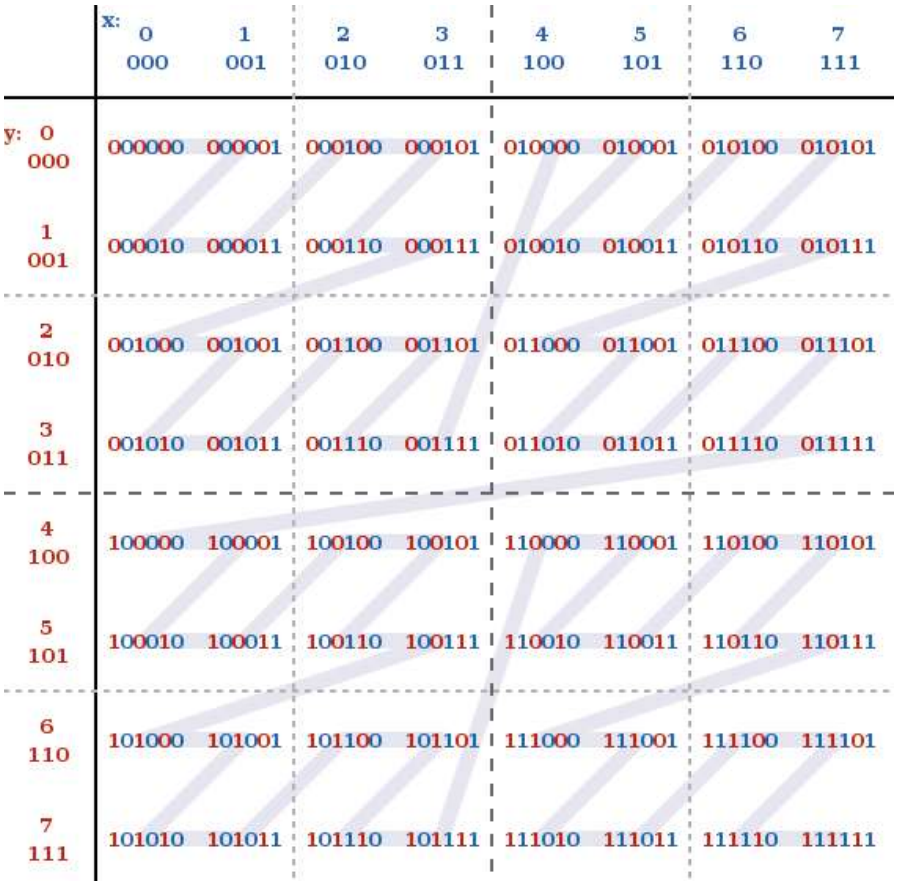

Figure 3.8: The 1D axis coordinate of the Z-order curve is the interleaved binary representation of its 2D coordinate values. - *Image by user Nomen4Omen for https://commons.wikimedia.org/.*

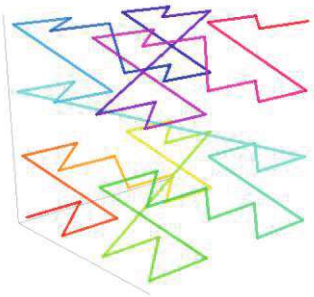

Figure 3.9: 3D Z-order curve - second iteration. - *Image by Robert Dickau for https://commons.wikimedia.org/.*

3.2 Databases and Spatial Indices

Relational Databases

Relational databases are frequently employed for managing spatial data, given their capacity to effectively store and process large datasets. Spatial indexing techniques can be integrated into relational databases to facilitate spatial queries and analysis. By minimizing the amount of data that needs to be scanned for a query, these spatial indexing structures can enhance the efficiency and precision of queries. Relational databases can leverage spatial indexing structures such as R-trees, quadtrees, and grid-based methods. Further, many relational database management systems, including MySQL, PostgreSQL, and Oracle, offer inherent support for spatial data types and spatial indexing techniques, simplifying the process for developers to store and query spatial data within a relational database environment.

Oracle's Spatial Database [62], IBM's Informix [38], the PostGIS [68] extension for PostgreSQL, and MySQL are all examples of data-driven approaches that use R-trees. PostGIS is able to handle 3D data like digital elevation models. Another extension named pgPointCloud enables the handling of point cloud data. IBM's DB2 introduces a new geodetic Voronoi index, which employs minimum bounding circles to create cells in a Voronoi tessellation of the Earth's surface, combined with an R-tree indexing the Voronoi diagram points [77]. Oracle's spatial grid index, Amazon Aurora MySQL, IBM DB2, and Microsoft SQL Server are examples of databases supporting space-driven 2D grid indexing approaches.

The data in this thesis's primary use cases, however, is 3D data and based on a voxel grid, which diverges from the standard 2D polygonal or grid data that relational database approaches are optimized to handle. Even though the data partially includes segmented objects, most of the SPX framework's implemented queries focus on raw image data and derived data.

NoSQL Databases

Recently, NoSQL databases such as MongoDB, Google Firestore, Apache Cassandra, Redis, and Azure Cosmos DB have been gaining popularity, largely due to the surge in user-generated unstructured data on social networks. Emerging paradigms gaining traction include object-oriented models, document-oriented databases, graph databases, and key-value stores. However, none of these NoSQL concepts are particularly well-suited for handling volumetric data, despite their increasing use in GIS research. While broad support for handling volumetric data is lacking in most industry-standard NoSQL databases, the following databases do offer support for geospatial data:

CouchDB is a document-based database that has a spatial index extension known as GeoCouch [56]. GeoCouch also employs an R-tree, but can only handle 2D data.

Azure Cosmos DB [54] is a globally distributed, multi-model database service that supports multiple data models, including document, key-value, graph, and column-family

data models. It also offers support for geospatial data, but specific implementation details are not readily available. The geospatial part of the database is designed for 2D data.

MongoDB [17] is a document-oriented database that supports geospatial data types and queries. It uses a geospatial index to support queries based on location, and distance. The geospatial part of MongoDB is designed for 2D data with limited support for an additional height value (2.5D).

SPX and Databases

While extensions for spatial data handling and indexing in Relational Database Management Systems (RDBMS) are available, representing complex data types for different use cases in a relational database can be challenging. In a NoSQL database, data types can be more flexible, but this often comes at the cost of query performance, especially for spatial queries. SPX is designed to be easily extensible for different data types and representations while providing a robust and high-performance query engine for spatial data.

In the projects where SPX is utilized, query results are typically calculated by aggregating the indexed data in different ways, which can pose a challenge when using RDBMS. This is because RDBMS have limited support for on-the-fly aggregation, often requiring the entire dataset to be fetched to perform the aggregation on the full result, leading to low query performance with increased memory overhead. This approach is impractical for our project's purposes, as the overhead could equate to several gigabytes of data required to be kept in-memory. Additionally, depending on the use case, SPX is able to avoid reading data that is unnecessary for the result calculation by filtering beforehand.

3.3 Neuronal Databases

A lot of work has been done on creating specialized databases to explore sample collections in the field of neuroscience. Koslow et al. [46] provide an overview of the techniques, tools, and issues of databasing neurological data. Schulze et al. [76] mention several examples. *FlyBrain* [5], which combines schematic representations, image atlases, and 3D object data, and *FlyBase* [83], which merges image, genetic, and molecular data with advanced semantic query support. *Virtual FlyBrain* [55, 63] builds upon FlyBase and enables the user to browse and annotate the 3D image stacks of FlyBase. Other notable neurological databases include:

CATMAID (Collaborative Annotation Toolkit for Massive Amounts of Image Data) [72] is an image resource designed for various biological specimens, and provides tools for collaborative annotating and analysis of large-scale image data, such as electron microscopy data. However, it does not support spatial indexing or query capabilities for volumetric data, nor does it support aggregation queries.

The **Allen Brain Atlas** [40] is a comprehensive resource that provides large-scale gene expression data and neuroanatomical information for the mouse and human brain. It

offers tools for interactive exploration and analysis of gene expression patterns. Spatial queries are limited to predefined regions, and analysis tools are available for the integrated datasets.

The **NeuroMorpho.org** [2] database provides a comprehensive collection of digitally reconstructed neurons. It supports the exploration and analysis of neuronal morphologies, facilitating comparisons across different species and brain regions. Researchers can submit new data for integration, but the database is limited to neuron morphology data. Users can search the integrated datasets using metadata, but spatial queries are not supported.

OpenNeuro [49] is a platform that hosts and shares neuroimaging data. It provides access to a wide range of brain imaging datasets in various modalities, including magnetic resonance imaging (MRI), positron emission tomography (PET), magnetoencephalography (MEG), and electroencephalography (EEG) data. It offers searches based on metadata and limited visualization capabilities. OpenNeuro acts primarily as a data repository and does not provide spatial indexing or query capabilities.

SPX and Neuronal Databases

While the aforementioned platforms provide semantic querying based on metadata, querying spatial data is either not possible or is limited to predefined regions. SPX is designed to handle large-scale spatial grid data queries, focusing on volumetric grid data and region-based data. Its goals are to provide an easy way to integrate new data abstractions and query derivative data, as well as to create new query types. Query results of SPX are typically supplemented by external resources, such as a RDBMS. SPX focuses on high-performance aggregation of range queries rather than on access and discovery of data according to metadata. Providing high-performance data access for predefined regions is also within the capabilities of SPX, leveraging its region-based approach.

3.4 Predecessor Indices

The SPX framework draws inspiration from several predecessor projects that utilized fixed grid indexing in combination with space-filling curves to manage spatial data. These projects focused on indexing and querying large-scale volumetric data derived from confocal light microscopy images. The resulting data structures, which provide indexed access to the stored data, are consistently referred to as an *index* across these projects.

3.4.1 Distance-Field Index

The earliest project was conducted by Bruckner et al.[12] and Solteszova et al.[79], which aimed to identify structures in the brain of the adult Drosophila melanogaster that overlapped with or were close to a region of interest. The indexed data consisted of signed distance fields representing objects within the indexed space. This enabled users to query for structures near or overlapping the region of interest.

3.4.2 Staining Index

Schulze et al. [76] employed fixed grid indexing on binary volume masks, which marked voxels with strong staining signals in the source images. The index was used to find GAL4/UAS staining images of the adult Drosophila melanogaster brain that exhibited strong staining within a specified region of interest. This query type was termed a *high-staining query*. The second type of query, the *similar-staining query*, identified images with staining patterns similar to those in a selected reference image within the query area.

3.4.3 Structure Index

Ganglberger et al. [26] indexed gradient vector flow fields to identify images in a GAL4/UAS staining image collection that shared similar structural properties with a given reference image within a query area. By utilizing gradient vector flow fields, the index could identify images with similar structures, even if they were spatially shifted. This type of query was referred to as a *similar-structure query*.

Summary

Data-driven spatial indexing methods, such as Binary Space Partitioning (BSP) trees, R-trees, and their derivatives, offer optimized and effective strategies for storing spatial data based on polygons. Many relational databases, including Oracle, IBM Informix, PostgreSQL, and MySQL, integrate these methods due to their built-in support for spatial data types and spatial indexing. However, these approaches are primarily tailored toward managing polygonal data, not volumetric data. Therefore, despite their strengths, they do not perfectly align with the data use cases of the SPX framework, which primarily deals with volumetric grid data.

Grid-based spatial indexing, as utilized in the predecessor prototype projects, organizes data in a fixed grid format, which is particularly suited for managing volumetric and voxel grid-based data. This approach capitalizes on the inherent structure of the data, making it especially useful for handling large-scale, unstructured data sets, such as those generated by confocal light microscopy images. The use of fixed cell grid indexing allows efficient management of area-based queries, primarily by maintaining the locality of the data on the storage medium to maximize cache hierarchy utilization and read-ahead caching. The indexing is accomplished by a space-filling curve mapping of the data, such as the Hilbert and Z-order curve.

Following the proven concept of the predecessor projects by Bruckner et al. [12], Solteszova et al. [79], Schulze et al. [76] and Ganglberger et al. [26], SPX utilizes a space-filling curve to index the reference space and registration volume for the primary use case of unstructured volumetric image data, but with extensions to support segmented structures and regional data.

CHAPTER 4

Key Terms

This chapter introduces the key terms and definitions used throughout this thesis, which are essential for understanding the detailed explanations of the methodology, as well as the development and implementation of the SPX framework. It serves as a general high-level explanation of the terms—a more detailed explanation can be found in the respective sections of the thesis.

SPX is a spatial indexing framework for *voxel grid data* and *region data*. It operates on built *indices*, and the encoding and decoding of the index data are performed by *codecs*, which implement the data handling, the supported data types, and query mechanisms of an index of its type.

4.1 The Reference Space

The *reference space* is the 3D coordinate space that SPX indexes. In cases where the indexed data is image data, the reference space is typically determined by the registration template. In other cases, the reference space serves as a common coordinate system to which data from various sources are mapped before the indexing process.

4.2 Index Items and Item Identifiers

Index items are the data sources for the index data, such as single-channel images containing GAL4/UAS staining. Depending on the codec used, the index data are either derived from the data of the index items or directly indexed. Each index item has a unique identifier to identify the data source of queried data. An item identifier has a certain structure, and throughout this thesis, the structure's name *ItemID* is used for this identifier. Typically, queries on the index data return the item identifiers and result values of the index items that match the query criteria.

4.3 Index Data, Data Entries, and Data Pages

The index data are the data that are indexed, stored, and used to answer queries. They consist of the *data entries* of the index items and can be either voxel grid data or region data. A *data page* describes a block of memory in which one or multiple data entries are stored.

Voxel Grid Data - Voxel Grid Indexing

Voxel grid data have a one-to-one mapping between the coordinates of the reference space and the data entries of the index items. During the creation process, each index item provides a data entry for each coordinate in the reference space. This data entry may be a nil value, indicating that the index item does not have data at this coordinate. In an index, each coordinate of the reference space has multiple data entries, but only one data entry per index item.

Region Data, Regions, and Region Samples - Region Indexing

The region data SPX handles have a many-to-many mapping between the coordinates of the reference space and the data entries of the index items. Multiple coordinates in the reference space (a *region*) can point to one or multiple data entries of a *region data collection*. Because the data entries belong to a region, they are referred to *region samples* throughout this thesis. Each coordinate can have multiple regions associated with it. The region indexing is a two-step mapping: first, the coordinate-to-regions mapping, and second, the region-to-samples mapping. Typically, the region data is a collection of region samples, grouped by regions. Therefore, throughout this thesis, the term *region data collection* is used to describe the data source of the region data.

4.4 Indices and Index Types

In the context of SPX, *indices* are instances of data structures that map coordinates to *data entries* of *index items*. The process of creating this mapping is called the *indexing*. Indices consist of a lookup component that performs the mapping and a data component that stores the index data in a retrieval-optimized manner. Each index is always of a specific index type, defining its purpose and data, such as a *staining index* for indexing GAL4/UAS *staining data*. Each index is also of a specific kind based on the kind of codec used. As the name implies, a *voxel codec index* uses a codec that works with voxel grid data, while a *region codec index* uses a codec that operates on region data. Currently not used, but a *hybrid codec index* is entirely possible with a codec that implements the voxel codec and the region codec interface.

4.5 Codecs and Queries

In SPX, each index is based on a codec; for example, a *staining index* [76] is based on the *staining codec*. In computer science, the term codec typically refers to a coder-decoder pair used for encoding and decoding data. Codecs encode the data entries of the index items during the indexing and decode them during the query execution. Following the two basic kinds of index data, a codec is either of kind *voxel codec*, *region codec*, or both. Codecs are implementations of the *voxel codec interface*, the *region codec interface*, or of both.

The Voxel Codec Definition

During the indexing process, the voxel codec is responsible for providing and encoding the data entry of each index item for each coordinate in the index space. As the name implies, the voxel codec operates on voxel grid data. The voxel codec also defines all the queries that can be executed on the voxel grid data in the index by providing the implementation of the different query result aggregations.

The Region Codec Definition

The region codec handles the data mapping required for the region data by providing the two-step mapping from *coordinates-to-regions* and *regions-to-samples* during the index creation process—the indexing of the reference space. It is also responsible for encoding the region samples of a region once they are requested by the index creation process. The region samples are the data entries in the index, while a region is only a set of coordinates to which the samples belong. The region codec also includes the decoding of the region samples and defines the queries that can be executed on the data.

Queries

Queries use a subset of the indexed data to aggregate a result. Since the codec defines the data type, a codec also defines the queries that can be executed on the index data. The queries are implemented by the codec, and every query is executed by an *query processor* instance instantiated by the codec, which calculates the query result by aggregating the data entries of the index items.

CHAPTER 5

Framework Parts & Structure

As outlined in Section 1.3, the SPX framework's goal is to generalize the spatial indexing approach used by Bruckner et al.[12], Solteszova et al.[79], Schulze et al.[76], and Ganglberger et al.[26]. The aim is to extend this approach and create a flexible, high-performance base for further developments, such as novel indices and queries that use voxel grid data. In addition to indexing voxel grid data, SPX should support the indexing of region data with multiple region samples in the indexed reference space.

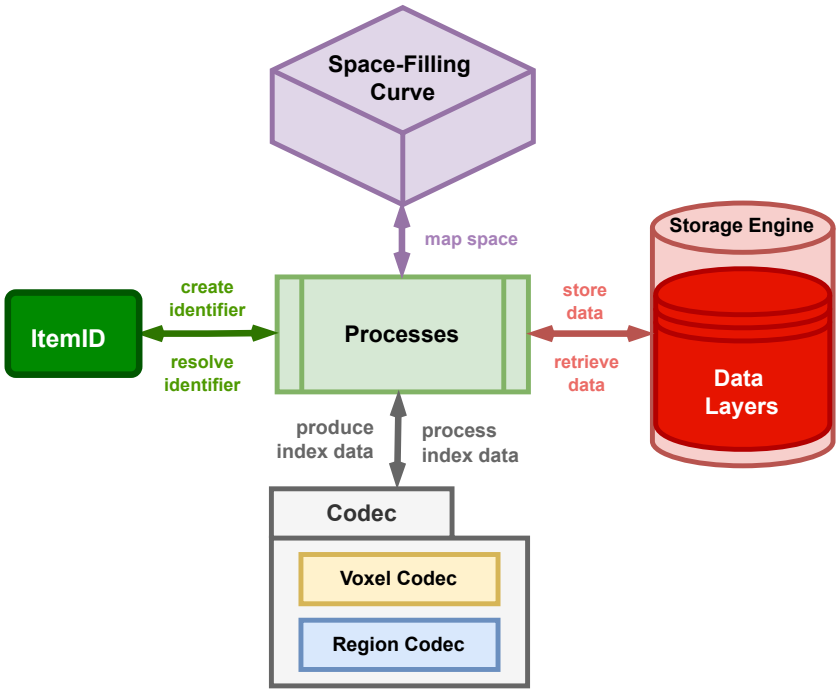

Figure 5.1: Conceptual overview showing the main parts of the SPX framework.

The SPX framework is built in an extensible fashion. Several parts of it are abstracted by interfaces. Specialized or new structural concepts can be implemented using these interfaces. The core of SPX, the query engine, does not need to be changed when a part is replaced or specialized. Figure 5.1 provides a general overview. The implementations of the colored parts can be replaced without interfering with other components by implementing their interface definitions. The parts are outlined in the following sections, starting with the index item identifier scheme *ItemID*.

5.1 Identifying Index Items - ItemID

Every index item is identified by a unique identifier—the *ItemID*. One of this project's goals was to extend the identifier scheme used by Schulze et al. [76], which involved only an integer, to a version containing additional information that supports the identification of structural types already at the index level. Furthermore, information about the dataset or project to which an index item belongs has been added. The resulting item identifier consists of: a *dataset ID*, a *structural type ID*, and a *database ID*. The user must provide this information for every index item. The structural types of ItemIDs are given by the usage context of SPX and consist of the following types:

Image Types

- *Single Channel Images*
- *Average Images*

Biological Types
Biological structural types originate from segmented neurological structures.

- *Area Types*: *Arborizations, Neuropils, Cell Bodies*
- *Tubular Types*: *Axon Tracts, Projections, Connections*

Region Indexing Types

- *Regions*
- *Region Samples*

Typically, voxel grid data is provided to SPX as an index item—a tuple of ItemID and index data source. In contrast, region data is handed over as a data collection consisting of regions and region samples. Both regions and region samples are identified by ItemIDs of the respective corresponding type. The ItemIDs for the regions and samples are generated during the indexing of the data collection, which requires that each region and

sample is uniquely identifiable within its respective region data collection. A new ItemID scheme can be integrated into SPX by providing an implementation of the interface shown in Listing 1.

```
type ItemID interface {
   // serialize the ItemID to a string
   ToString() string

   // the TypeID is a unique identifier for the structural type of an ItemID
   // all possible TypeIDs are defined in an enum (a byte)
   GetTypeID() TypeID

   // get the dataset ID as a string
   // (it can be of any string-serializable type internally)
   GetDatasetID() string

   Equal(ItemID) bool

   // the actual ID can be of any type, but it must be serializable and
   ↪  comparable
   ID any
}
```

Listing 1: The interface a new ItemID scheme has to implement. - *in Go*

5.2 Space-Filling Curves in SPX

The space-filling curve is a crucial part of the SPX framework. It maps the 3D coordinate space of the reference space to a 1D indexing space. The space-filling curve is used to create a unique 1D index for each coordinate in the 3D reference space. The indexing happens along the space-filling curve's mapping of the reference space, and therefore the space-filling curve defines the data order in the index. The curve's properties, such as locality preservation, are essential for the performance of the index.

A Morton curve, or Z-order curve, explained in Section 3.1.3, is employed as the default space-filling curve. This choice is due to its straightforward implementation, well-defined nature in 3D, and superior performance in the mapping and reverse mapping of the 3D voxel coordinates to the 1D space-filling curve coordinate, also termed the *space-filling curve index*. Although the Hilbert curve, utilized in predecessor projects, potentially offers better locality preservation and reduces cache miss risk, it demands more computational resources to calculate and is not well-defined in three-dimensional space, as Haverkort states [34]. Depending on the scenario, other indexing strategies using different space-filling curves might improve the effectiveness of a computer's cache hierarchy. Therefore, SPX offers the possibility to implement new space-filling curves by implementing its space-filling curve interface shown in Listing 2.

```go
// initializes a space-filling curve of the given type
func InitSFC(sfcType ESpaceFillingCurveType, maxX, maxY, maxZ int) SFC

// the space-filling curve interface definition
type SFC interface {
   // map 3D coordinates to 1D index
   Pos3DToIdx(x, y, z int) uint64

   // map 1D index to 3D coordinates
   IdxToPos3D(idx uint64) (x, y, z int)

   // get the last 1D traversal curve coordinate of the reference space
   GetMaxIndex() uint64
}
```

Listing 2: The space-filling curve interface must be implemented by any space-filling curve used within the framework. - *in Go*

The cache hierarchy comprises a sequence of memory caches in a computer, ordered into several levels to boost the system's performance by reducing the average data access time [75]. This structure includes layers of caches such as the L1 cache, which is the smallest and fastest and is situated closest to the processor, and the L2 and L3 caches, which are also integrated into the CPU. Beyond these, there's the RAM, usually followed by the hard drive. Although there exist configurations with additional caching layers like a RAM disk or Intel Optane, this structure is the most common. Each level of cache in the hierarchy is intended to hold a subset of the data stored in the cache of the next lower level, with the goal of storing the most frequently accessed and most likely to be accessed data in the fastest and smallest caches. When a processor requests data that is not in the cache, it triggers a cache miss, and the data is obtained from a lower and slower cache level, or ultimately from the storage media. Read-ahead prefetching is a general optimization technique used to enhance cache performance. When the cache controller recognizes that a specific data block is being accessed, it tries to anticipate the next blocks to retrieve in advance before it is requested. This helps to minimize cache misses, improves the overall system latency, and happens on all caching layers. Each hard drive possesses its own cache memory and controller. By ordering data according to a space-filling curve, the data is stored to maximize spatial coherence within the cache hierarchy, to increase the probability of data being prefetched into a higher layer's cache before it is requested.

5.3 Codecs Explained

One requirement for this project was the integration of new data types and concepts, to support new types of queries. Codecs are the combination of a data type, and queries implemented on it. It is responsible for providing the index data entry (encoding) during creation, and the index data entry processing, including the decoding, during a query result aggregation. Schulze et al.[76] described two types of spatial indices, hence two codecs: the *staining index* and the *distance-field index*, already used by Solteszova et al.[79] and Bruckner et al.[12] before. Ganglberger et al.[26] also preprocessed and compared the same kind of voxel grid data; therefore, their *structure index* was added as a third codec. Subsequently, in the context of BrainTrawler[25], the *gene-expression-value index* was implemented to index gene expression data mapped onto a reference space.

All the codecs mentioned above are used in voxel grid indices, meaning that they provide a data entry for each index item at each coordinate. Additionally, the requirement for region indices was established due to the necessity to query single-cell RNA samples representing brain regions. Access of those region samples should happen according to a query area in the reference space overlapping with brain regions, or by using brain regions directly. An index for indexing single-cell RNA sample data is the *gene-sample-meta index*, which is shortly described in the context of its using applications [25, 27] in the Applications Chapter 8.

The indexing of voxel grid data and region data are different approaches. While simple support of regions with only one data entry requires only a change to the data lookup part, the requirement to have multiple region samples necessitates the mapping of regions to region samples. This additional mapping is required to support several thousand region samples, with possibly large payloads. Each sample has to be indexed separately and may have a different size due to the support of complex, possibly large data. To achieve the required flexibility and reflect the distinct concepts of voxels and regions, two separate interfaces were designed, providing the necessary abstraction between index data encoding and decoding, data handling, and query processing. One interface defines the requirements for a *voxel codec*, and the other for a *region codec*. Each codec can implement one or both interfaces, depending on the requirements and use cases of the codec and the indices based on it. Each of these basic interface definitions is again split into two integral parts, mirroring the two general stages of an index: the *data factory* used during the creation of an index, which is responsible for delivering the index data, and the *query processor*, which processes and aggregates the index data during a query. In the following sections, the interfaces are described in detail.

5.3.1 The Voxel Codec Interface

Listing 3 shows the interface definition of the voxel codecs. Voxel codecs for indices based on voxel grid data utilize fixed-size data entries, which simplifies the unpacking of data entries associated with a coordinate. The voxel codec interface includes the setup function definition for the *data factories*, used during the index creation process, as well as the setup function definition to create *query processors*, which contain the logic of the query result aggregation. In both parts of the interface, a factory pattern [23] is employed: the codec creates a data factory for each index item and a query processor for each query. Both are described in the following sections.

```go
type VoxelCodec interface {
    // the size of one index entry in bytes
    GetPayloadSize() int

    // initialize a data factory for an index item with the given data source
    ↪  path
    SetupVoxelDataItemEntryFactory(
        dataSource string,
        id ItemID,
        indexDimensions []int,
        layers int
    ) (VoxelDataEntryFactory, error)

    // initialize a query processor for a query type with the given
    ↪  parameters
    GetVoxelQueryMethod(
        queryName string,
        layers int,
        params []string
    ) (VoxelResultMethod, error)
}
```

Listing 3: The voxel codec interface, showing the methods for setting up data factories and query processors. - *in Go*

Voxel Data Factories

A voxel codec's implementation requires a method to generate data entries for each index item at every coordinate. To accomplish this, the index creation process initializes a data factory for each index item, whose interface definition is shown in Listing 4. These factories are responsible for retrieving and encoding the data entries from their respective sources. Generating a data page for one coordinate involves iterating through all data factories—and thus all index items—to obtain the corresponding data entries. If a data entry at the coordinate is nil, the index item is ignored. The data page is assembled from all non-nil data entries in combination with their respective ItemIDs—the index

item identifiers. The storage engine stores the number of index items that have a data entry at a given coordinate. This information is crucial for efficiently disassembling a data page into ItemIDs and data entries.

```go
type VoxelDataEntryFactory interface {
   // retrieve the data entry for a given coordinate of the factory's index
   ↪  item
   GetItemDataEntry(layer int, x int, y int, z int) bytes.Buffer
}
```

Listing 4: The data factory interface for a voxel codec is responsible for generating data entries for each index item. - *in Go*

Voxel Query Processors

In the context of the voxel codecs, a query aggregates the data entries for each queried coordinate. Each query is identified by a specific name and may require parameters. A voxel codec's query processor processes the data entries of the index items at a queried coordinate and aggregates the query results. Listing 5 shows the interface definition for a query processor.

```go
// one single index item identifier with its data entry of one coordinate
type VoxelEntry struct {
   ItemID MappedID
   Payload  []byte
}

// the query processor interface of the voxel codec
type VoxelResultMethod interface {

   // handle the data entries at a coordinate
   HandleCoordinateResult(
      readData []VoxelEntry,
      position Position,
   ) bool

   // get the query results
   // "any" describes that any type of data can be returned
   GetResults() any
}
```

Listing 5: The interface definitions for the query processors of a voxel codec. - *in Go*

5.3.2 The Region Codec Interface

The region codec interface also includes the index *data factory* and the *query processor* interface. Listing 6 shows the high-level interface definition. Unlike voxel codecs, which use fixed-size data entries, the region codec must support complex data entries of variable sizes.

```
type RegionCodec interface {
   // initialize the data factory for an index item with the given data
   ↪  source path and the index item's name
   SetupRegionDataItemEntryFactory(
      dataSource string,
      regionCollectionName string,
      indexDimensions []int,
      originalDimensions []int
   ) (RegionDataEntryFactory, error)

    // encode the data entry of a region sample to a data page
   // "any" describes that any type of data can be returned
    EncodeRegionSampleDataEntry(
      itemToEncode any,
      layer int
   ) (encodedDataBuf *bytes.Buffer, err error)

   // decode the data entry of a region sample from a data page
    DecodeRegionSampleDataEntry(
      rdr io.Reader,
      layer int,
   ) (any, error)

   // initialize the query processor for a query with the given parameters
    GetRegionQueryMethod(
      queryName string,
      layers int,
      params []string
   ) (RegionResultMethod, error)
}
```

Listing 6: The interface definition of the region codec showing the instantiation methods for the region data factories and query processors. - *in Go*

Region Data Factories

The index creation process for region data generates a data factory for each region data collection. Listing 7 shows the data factory interface for the region codecs. Unlike voxel codecs, where one data factory produces a single data entry for each index item at a given coordinate, a region codec's data factory produces several data entries, each identified by its own ItemID identifier. In a region codec, multiple coordinates are assigned to

one or more regions, and each region may contain several region samples. Consequently, the data factory in a region codec is responsible for mapping coordinates to regions and regions to region samples. Each collection typically requires the loading several data sources, including these two mappings and the data sources for the region samples. In addition to providing these mappings, the region codec's data factory creates the data entries for each region sample. Due to the requirement to support complex data entries of variable size, the region codec has an explicit encoding and decoding method for the data entries of the region samples.

```
RegionID = ItemID with type: region
SampleID = ItemID with type: sample
// "any" describes that any type of data can be returned
any = a decoded region sample item

type RegionDataEntryFactory interface {
   // get the region ItemIDs at a coordinate
   GetRegionIDsAtCoordinate(x, y, z int) []RegionItemID

   // get the region sample ItemIDs of a region
   GetRegionSampleIDs(regionID RegionItemID) []SampleItemID

   // get the data entry of a region sample
   // the encoding of the data entry is handled by a separate region codec
   ↪  method
   GetRegionSampleItem(sampleID SampleItemID, layer int) any
}
```

Listing 7: The interface definition for the region codec's data factory. The encoded sample data type is only known to the region codec, but not at the interface definition. Any allows for arbitrary types to be used in Go. - *in Go*

Region Query Processors

Listing 6 shows the interface a region query processor has to implement. As in the voxel codecs, queries are identified by their names and parameters, with which a query processor is initialized by the codec for each query. A region query processor has filter methods, which are utilized to filter the queried regions and region samples before reading the sample data entries. This is to avoid the unnecessary reading of potentially large data.

```
RegionID = ItemID with type: region
SampleID = ItemID with type: sample
// "any" describes that any type of data can be returned
any = a decoded region sample item

type RegionResultMethod interface {

   // to filter queried regions and region samples before reading the sample
   ↪  data entries
   RegionFilter(regionIDs RegionID) bool
   SampleFilter(sampleID SampleID, regionID RegionID) bool

   // handle the data of one region sample
   HandleRegionSampleItem(
      sampleID SampleID,
      decodedRegionSampleItem any,
      regionID RegionID
   ) bool

   // get the query results
   GetResults() any
}
```

Listing 8: The interface definition for a query processor of a region codec. The decoded sample data type is only known to the region codec, but not at the interface definition. Any allows for arbitrary types to be used in Go. - *in Go*

5.4 Data Layers

Data layers can be viewed as distinct partitions designed to store multiple data entries for a single index item at the same coordinate in voxel codecs. In region codecs, they store multiple data entries for the same single region sample. The data in each layer is independent of other layers, and the layers follow a demand-based access principle. This means that a layer is accessed if a query cannot be adequately resolved using data from preceding layers. For voxel codecs, each layer consists of all the data pages for all coordinates, while for region codecs, each layer contains all the data pages for all region samples. The concept of data layers is designed to support multiple data types within a single index, enabling the aggregation of different data types or the creation of hierarchical queries.

5.5 Storage Engines in SPX

In the context of databases, a storage engine refers to the underlying software component of a database management system that handles storage, retrieval, and access. It acts as the low-level interface between the database management system and the data storage. In SPX, the storage engine is responsible for managing data storage and indexing structures,

ensuring data integrity, and processing read and write requests. Different storage engines in SPX can offer varying performance characteristics, making them better suited to different use cases. The storage engine is divided into two parts resembling the two basic codec types: the *voxel part* and the *region part*. The voxel part is responsible for handling voxel grid data, while the region part manages region data. The storage engine interface is defined for both parts, as shown in Listings 9 and 10.

```
// type definitions for the interface
type Position struct { X, Y, Z int  }

type VoxelRequest struct {
   Position              Position
   Position1D            uint64
   Layer                 int
}

type VoxelData struct {
   Position              Position
   Position1D            uint64
   Layer                 int
   numEntries            int
   Data                  []byte
}

// the storage engine interface for the voxel grid data
type IStoreVoxel interface {

   // provide a voxel request channel, and receive a response channel of
   ↪  the retrieved voxel data
   GetVoxelData(toRead <-chan VoxelRequest) <-chan VoxelData

   // during indexings, this method sets the encoded data entries for a
   ↪  coordinate in the storage engine
   SetVoxelData(
      layer int,
      pos Position,
      pos1D int,
      numEntries int,
      data []byte
   )
}
```

Listing 9: The interface for managing voxel grid data storage, including methods for retrieving and setting data. - *in Go*

```go
type RegionID  ItemID  with type: region
type SampleID  ItemID  with type: sample

type SampleInformation struct {
   IndexItemID string
   RegionID  RegionID
   SampleID  SampleID
}

// sample data structure expands sample information
type SampleData struct {
   SampleInformation
   Layer      int
   Data       []byte
}

type SampleToRead struct {
   SampleInformation
   Layer int
}

// reference space coordinate and 1D space-filling curve index
type CoordPair struct {
   Pos3D Position
   Pos1D int
}

// the storage engine interface for the region data
type IStoreRegion interface {
   // gets the regions of a coordinate
   GetRegionsOfCoordinates(pos []CoordPair) []RegionID
   // sets the regions of a coordinate during the indexing process
   SetRegionsOfCoordinate(p3D Position, p1D int, regionIDs []RegionID)

   // gets the encoded region samples of a region
   GetRegionSamples(regionID []RegionID) []SampleInformation
   // sets the encoded region samples during the indexing process
   SetRegionSampleData(
      layer int,
      regionID RegionID,
      sampleID SampleID,
      sampleData []byte
   )

   // provide a sample request channel and receive a response data
   ↪  channel
   GetRegionSampleData(<-chan SampleToRead) <-chan SampleData
}
```

Listing 10: The interface for managing region data storage, detailing the methods for handling region samples and their associated data. - *in Go*

One of the fundamental concepts of the storage engine in SPX is the use of Go's channels [67] to handle requests and responses. The storage engine uses a buffered channel to receive requests and one to send retrieved data pages as responses. Due to the buffered nature of the channels, the storage engine can receive data requests while it is busy fetching data, and can insert fetched data into the response channel even though SPX is busy handling data responses from previous requests. Data fetching by the storage engine is therefore done asynchronously, and independently from the data processing, by a separate reader Go routine.

While a voxel codec only requires a coordinate-to-data-page lookup structure internally, the query handling of a region codec necessitates the aforementioned two-step mapping to evaluate which region samples are relevant for a query. This two-step mapping also enables direct access to data pages based on requested regions and region samples, bypassing the coordinate-to-regions mapping. The lookup structures for both, voxel codecs and region codecs, are kept in memory to avoid read-time delays from the storage media. In both cases, the storage engine and the data page lookups must support the multilayer approach described in Section 5.4. *Loading* or *opening* an index means that the storage engine loads the lookup structures into memory from their serialized form on the storage media, and prepares to access the index's data pages by opening the buffered request and response channels.

CHAPTER 6

Methodology

This chapter gives a description of the methods, concepts, and solutions used to address the use cases that the SPX framework aims to cover. It starts with a temporal overview of the development process and workflow, followed by a discussion of spatial indexing in the context of the two fundamental data types that SPX handles: voxel grid data and region data. After that, the creation and querying of indices are discussed.

6.1 Development Process and Workflow

This section gives an overview of the development process and workflow, and acts as a short summary of the development of the SPX framework. All implemented features, codecs, and parts are described in detail later in this thesis. The development of the SPX framework occurred in incremental steps. After establishing the common requirements for the voxel grid data, the framework development began with a prototype by Schulze, based on the staining index [76]. Unlike the description in the technical report [76], the prototype employed a Z-order curve as the space-filling indexing curve. The prototype was then used to create a staining index for the data available at that time within the Larvalbrain project [86].

Development for Voxel Grid Data

The Larvalbrain project [86] aims to serve as a resource for exploring Drosophila melanogaster split line samples. At the project's beginning, a collection of 5008 split line GAL4/UAS sample images at the L3 larval stage was available, registered on a newly created template using Larvalign [60]. I used the registered single-channel images of the anti-GFP channel throughout the development of the SPX framework to derive index data for several indices created by SPX. In addition to the GAL4/UAS confocal microscopy sample images, segmented objects were available as polygonal meshes in the case of neuropils and arborizations, and axon tracts as skeleton graph files.

The GAL4/UAS single-channel images from the Larvalbrain project were preprocessed into binary staining masks, as described by Schulze et al.[76], and used to create a staining index with the prototype at the start of the project. This staining index and the preprocessed binary staining masks formed the foundation for the initial SPX development.

Voxel Indices and Codecs

To accommodate different index types, I created an abstraction layer by defining an interface between image data handling and the staining index data-specific parts. This interface established the implementation requirements for all voxel grid indices and is referred to as the *voxel codec interface* throughout this thesis. To facilitate the implementation of codecs and enable in-memory indices, I introduced the *storage engine* concept, which handles data access and storage. Storing index data directly in memory allowed faster access times, easier debugging, and simpler testing of the data in an index. Additionally, serializing the index data in JSON format enabled both restoring in-memory index data and saving it in a human-readable format.

After implementing the staining index, including the *high-staining query* and the *similar-staining query* as described by Schulze et al.[76], I introduced a *memory-mapped index file storage engine* to support larger indices. To verify the correctness of the prototype's query results, I manually calculated the results of both query types and used these as a baseline. This baseline was then used to validate the high-staining and similar-staining query results in the staining codec implementation using the voxel codec abstraction of the SPX framework. Queries were executed on both the prototype's staining index with its query implementation and on a staining index created by the SPX framework. This validation process, alongside unit tests and query pipeline tests, ensured that no errors were introduced during SPX development.

To support the inclusion of segmented objects in a staining index, segmentations were added as areas of high staining. This required a method to differentiate between types of indexed data, as up to this point, only the single-channel GAL4/UAS images were indexed using the staining codec. While the prototype implementation used only integer values to identify the indexed confocal microscopy images, I extended this by introducing an index item identifier structure called *ItemID*, which encodes the type of indexed data as well as a dataset or project identifier.

In the next step, I implemented support for index data consisting of gradient vector flow fields, and the *similar-structure query* introduced by Ganglberger et al. [26]. This implementation, named the *structure codec*, was integrated into SPX as a voxel codec following the voxel codec interface definition and is used to create *structure indices*. The similar-structure query results from a structure index of the GAL4/UAS in Larvalbrain were evaluated by expert users, by reviewing the top result images when querying for fine structures.

The separating of the data reading from query processing led to the development of the multithreading architecture for the SPX framework's *query engine*. The final voxel codec required by the Larvalbrain project was the *distance-field codec*, introduced by Solteszova et al. [79], featuring the *object query* and the *object-overlap query*. In addition to standard tests, a staining index containing object segmentations was used to validate the object query results of the distance-field index. This was done by verifying that the sample segmentations overlapping the query area were also found in the query results of a high-staining query on the staining index.
For the BrainTrawler project [25, 27], which required a way to spatially query gene expression data mapped onto a common frame of reference, Florian Ganglberger and I implemented the *gene-expression-value codec* as a voxel codec with the *average gene expression query*. Florian Ganglberger validated the test query results by comparing the calculated averages with those computed in R.

Development for Region Data

The BrainTrawler project serves as a resource for neurobiologists to explore brain region connectivity and gene expression data mapped onto a common reference space. These data include several published datasets of single-cell transcriptomics, consisting of sample meta-information and gene expression profiles from RNA sequencing. Due to the lack of spatial coordinates, the single-cell samples are assigned to brain regions. In this context, the index data consists of information from several CSV files containing a collection of sample data, which includes a sample's meta-information and gene expression profile. The gene expression profile represents the strength of gene expression measured in the sample for many observed genes. The nature of the data, consisting of the sample's metadata and gene expression profiles valid for all coordinates of a brain region, prohibited the use of the voxel grid indexing approach due to the large storage requirements for each voxel within a brain region. Also, BrainTrawler required ways to query the sample meta-information and gene expression profiles not only by a query area but by using the brain regions or sample identifiers directly.

The Region Codec

To support a collection of samples belonging to the same brain region, including the reference space coordinate mapping, I developed the concept of region codecs, defined by a region codec interface, as described in Section 5.3.2, along with necessary extensions to the query engine and available storage engines. To avoid reading the potentially large gene expression data of region samples not required in the result subset, I introduced the data layer approach, separating gene expression profiles from sample meta-information. This approach is implemented in both the voxel and region data handling of SPX, as well as in the query engine.

The encoding, decoding, and querying of the single-cell sample metadata and gene expression profiles was implemented in a region codec called the *gene-sample-meta codec*. This codec provides a query to average gene expressions over user-defined categories based

on the metadata of samples in a query area. It was used to create a *gene-sample-meta index* for the BrainTrawler project [25, 27] and was validated by comparing query results with those calculated in R by Florian Ganglberger.

Testing and Validation

Each codec was validated through a series of tests, starting with separately testing the implemented queries using small examples and comparing query results to precalculated results. The query engine and storage engine implementations were separately tested using a simple test codec that implemented the voxel and region codec interfaces. Index generation, merging, converting, and querying were tested with this test codec, as well as synthetic staining images to create a staining index. Integration tests, which involved creating test indices and querying them, were used to verify the executables of the SPX framework. At the end of the project, the overall performance was measured, improved, and evaluated using a combination of individual component tests and query pipeline tests, using several codecs and queries, as described in Chapter 9. This included evaluating the impact of space-filling curve indexing on modern hardware and identifying implementation bottlenecks.

6.2 Spatial Indexing Applied

SPX enables queries on a discrete Euclidean 3D coordinate space $V := (x, y, z)_{x=0,\dots,w;\ y=0,\dots,h;\ z=0,\dots,d}$ with width w, height h, and depth d. This space is called *reference space* or *reference coordinate space*. V is generally defined by the template to which all data to be inserted into the index is aligned. Elements $\vec{v} \in V$ are called voxel positions or simply voxels. In our use cases, reference spaces are generally resampled to isotropic voxel spacing, but this is not a necessary requirement. The explanations and definitions in this section describe a file-based approach, where the index data resides in a single file.

Queries typically involve a set of voxel positions $\{\vec{v} \mid \vec{v} \in V\}$ that are spatially close to each other. Thus, it is essential to maintain spatial coherence when storing index data to maximize the efficiency of the caching hierarchy and read-ahead prefetching. For this purpose, a space-filling curve is employed to order the information stored in the index files. The curve defines a mapping $s : V \mapsto L$ that linearizes V to the 1D indexing coordinates $L := (l)_{l=1\dots(w*h*d)}$, as described in Section 3.1.3.

6.2.1 Voxel Grid Indexing

The first type of index data SPX can handle is 3D voxel grid data aligned to the reference coordinate space V. Index items are generally 3D binary masks, 3D imaging or vector data files, having a data entry at each grid coordinate.

Let I denote the set of all index items i, and $i(\vec{v})$ the value (*data entry*) of index item i at reference space coordinate $\vec{v}$. $|i(\vec{v})|$ refers in the following to the byte size of the

datatype of indexed values at voxel level. Index items may have a value at a reference space coordinate that indicates no signal (a NaN-value). In this case, the index item is ignored for this coordinate. We let $n_I(\vec{v})$ denote the absolute number of index items at coordinate $\vec{v}$ which have a non-NaN value.

The voxel codec index $\mathcal{I}_{\text{vox}} := (\mathcal{F}_{\text{vox}}, \mathcal{L}_{\text{vox}})$ consists of two elements, the lookup table $\mathcal{L}_{\text{vox}}$ and the index file $\mathcal{F}_{\text{vox}}$. The index file contains for each $\vec{v} \in V$ a block $b_{\vec{v}}$ (data page) of indexed data entries $i(\vec{v})$ of index items $i \in I$, having non-NaN values, in the form of tuples $(i, i(\vec{v}))$ and

$$b_{\vec{v}} := (i, i(\vec{v}))_{i \in I \wedge i(\vec{v}) \neq \text{NaN}}$$

These blocks of tuples are linearly ordered according to the mapping provided by s and the structure of the index file is defined as follows:

$$\mathcal{F}_{\text{vox}}(V, s, I) := (b_{s^{-1}(l)})_{l=1\ldots(w*h*d)}$$

The lookup table $\mathcal{L}_{\text{vox}}$ is a lean 3D data structure defined by the reference coordinate space and contains for each coordinate $\vec{v}$ the necessary access information of a coordinate's data page, such as the number of index items with value, the starting offset for the data page block $b_{\vec{v}}$ in the index file, and the data page size to access and efficiently handle related data the index file $\mathcal{F}$.

$$\mathcal{L}_{\text{vox}}(V, s, I) := (\vec{v} := (x, y, z), n_I(\vec{v}), pos_{\mathcal{F}}(b_{\vec{v}}), |b_{\vec{v}}|)_{x=0,\ldots,w;\ y=0,\ldots,h;\ z=0,\ldots,d}$$

where

$$|b_{\vec{v}}| := n_I(\vec{v}) \cdot |i(\vec{v})|$$

denotes the byte size of data page $b_{\vec{v}}$ and

$$pos_{\mathcal{F}}(b_{\vec{v}}) := \sum_{\alpha=1\ldots(s(\vec{v})-1)} |b_{s^{-1}(\alpha)}|$$

denotes the starting position of $b_{\vec{v}}$ in $\mathcal{F}_{\text{vox}}$.

In a straightforward implementation, storing the data page size in the $\mathcal{L}_{\text{vox}}$ is redundant. However, to enable optimizations, such as compression (see Section 7.4) this information is required, as the data page size might change.

When accessing the voxel codec index with a set of query coordinates $\{\vec{v} \mid \vec{v} \in V\}$, one can establish an optimized access order in two ways. Either by retrieving the access information for the data pages from the lookup table $\mathcal{L}_{\text{vox}}$ and then ordering the access information according to the file offsets, or by employing the space-filling curve mapping s to the query coordinates, ordering them according to the mapped space-filling curve indices, and then accessing the access information in the lookup table $\mathcal{L}_{\text{vox}}$ in the same order.
Using the space-filling curve for the ordering of the query coordinates alleviates the need

to retrieve all the access information beforehand. The ordering along the space-filling curve effectively results in the same optimized order as the data pages are saved in the storage engine. The access information is then used to access the data pages in the index file $\mathcal{F}_{\text{vox}}$ in a serial manner, exploiting the fact that queries are often done on connected sets of coordinates with spatial proximity. Figure 6.1 illustrates the voxel grid indexing in the SPX framework in 2D.

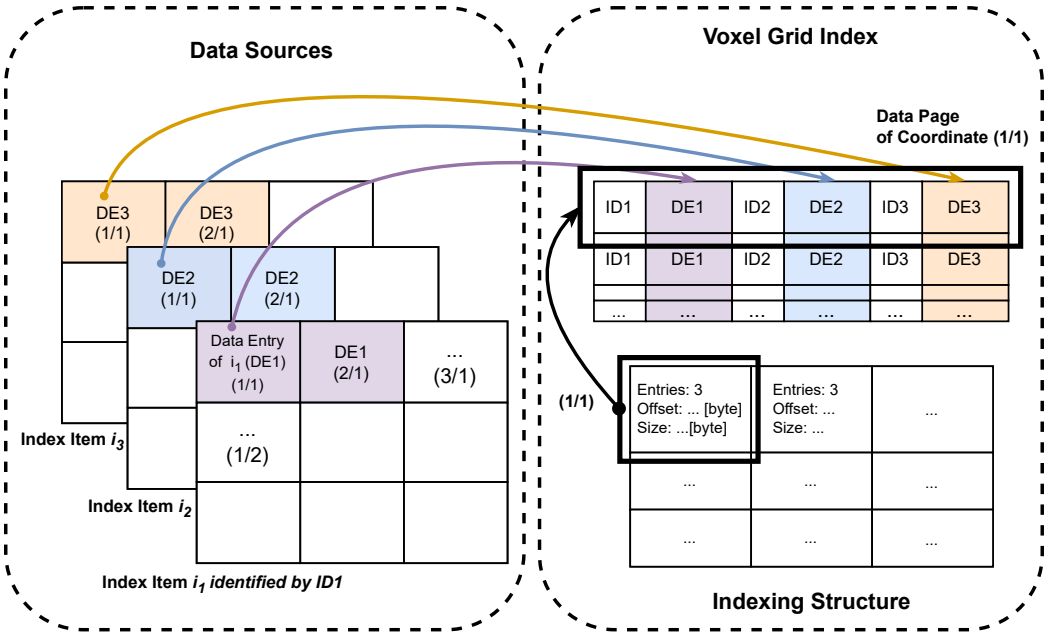

Figure 6.1: Voxel grid indexing in the SPX framework. $\mathcal{L}_{\text{vox}}$ is in this example a 2D structure containing the access information for each coordinate.

The usage of a 3D data structure to store access information stems from the fact that most space-filling curve mappings operate most efficiently on a cubic reference space, where the side length is a power of two. This often results in a much larger-than-necessary 1D lookup structure due to the padding required for the space-filling curve's mapping, leading to a higher memory footprint. Additionally, using a 3D structure inherently enables out-of-bounds checks.

6.2.2 Region Indexing

Region indexing is employed for datasets with highly complex and multimodal data including single-cell biopsy site or spatial transcriptomics data, that contain vast amounts of genetic data in combination with other derived information, like information on cell types, as well as experimental, clinical, and/or demographic information. A region r is a fixed, predefined set of coordinates: $r := \{\vec{v} \mid \vec{v} \in V\} \in R$, where R denotes a set of indexed regions of cardinality n_R. Inside the SPX framework, a region is identified by an ItemID of type *region*.

Again, a space-filling curve is used to establish a linear order of spatial regions, similar to the voxel index: by ordering all regions $r \in R$ by their first overlap with the space-filling curve, neighboring regions and in particular nested, hierarchically organized regions, are kept close together. The mapping defining the mapping of r to the position i in this ordered is defined by $l_R(r) := i \in \{1, ..., n_R\}$.

The region codec index $\mathcal{I}_{\text{reg}} := (\mathcal{F}_{\text{reg}}, \mathcal{L}^1_{\text{reg}}, \mathcal{L}^2_{\text{reg}})$ consists of index file $\mathcal{F}_{\text{reg}}$ and two look up tables $\mathcal{L}^i_{\text{reg}}, i = 1, ...2$.

Ingested region-wise data I is stored in the index file $\mathcal{F}_{\text{reg}}$ in blocks b_r containing the data related to a specific region $r \in R$. These data consist of an assembly of m_r region samples $s^j_r, j = 1...m_r$, each harboring a corresponding data entry p^j_r supporting complex data like sample metadata or a gene expression profile. This requires for each data entry of each region sample to be stored in its own data page within b_r and $b_r := (p^j_r)_{j=1...m_r}$.
Each region sample is identified by an ItemID of type *sample*.
The region blocks are ordered using mapping l_R to optimize the data access during queries. Thus

$$\mathcal{F}_{\text{reg}} := (b_{\tilde{r}_i})_{i=1...n_R} \text{ ,where } \tilde{r}_i := l_R^{-1}(i)$$

.

Two lookup tables support data access and complete the index:

Each coordinate $\vec{v}$ may belong to multiple regions. Lookup table

$$\mathcal{L}^1_{\text{reg}} := (\vec{v} := (x, y, z), R_{\vec{v}})_{x=0,...,w;\ y=0,...,h;\ z=0,...,d}$$

creates a mapping m from voxel position $\vec{v} \in V$ to the set of regions $R_{\vec{v}} \subseteq R$ overlapping with this position: $m(\vec{v}) := R_{\vec{v}} = \{r \mid \vec{v} \in r \wedge r \in R\}$. In this thesis, m is referred to as the *coordinate-to-regions mapping*.

Each region $r \in R$ relates to a set of region samples $S_r := \{(s^j_r) \mid j = 1...m_r\}$ that belong exclusively to this region and which contain the data entries of a region. Lookup table

$$\mathcal{L}^2_{\text{reg}} := (\tilde{r}_i, S_{\tilde{r}_i})_{i=1...n_R}$$

with $\tilde{r}_i := l_R^{-1}(i)$, creates a mapping $n(r) := S_r$ from a region $r \in R$ to an ordered set of region samples $S_{\tilde{r}}$, including the necessary access information for the data pages of the samples in the index file $\mathcal{F}_{\text{reg}}$, such as index file offset and data page size. The mapping n is also called the *region-to-samples mapping*.

To get access information for a set of region samples corresponding to one coordinate $\vec{v} \in V$, both mappings m and n have to be applied. To obtain the set of region samples for a query area spanning multiple coordinates, the set of regions for the second mapping

n is the unique set of all the regions found by applying the first mapping m to all coordinates of the query area. Since queried coordinates are typically close to each other, the combined set of regions for all coordinates includes many duplicates, which must be removed. After applying the mapping n to the unique set of regions, the region sample access information becomes available, and the data pages for the sample data entries can be fetched from the index file $\mathcal{F}_{\text{reg}}$. By ordering the query coordinates according to their mapped space-filling curve indices after applying the mapping s, the same order is achieved as the region samples stored in the storage engine, and the region ItemIDs are inserted into the lookup table $\mathcal{L}^2_{\text{reg}}$.

$\mathcal{L}^1_{\text{reg}}$ is a lean 3D data structure defined by the reference space. It contains, for each voxel $\vec{v} \in V$, the list of region ItemIDs $R_{\vec{v}}$. The second lookup table, $\mathcal{L}^2_{\text{reg}}$, is a hashmap where the region ItemIDs are the keys, and the region sample information—consisting of the sample ItemID and the necessary access information—are the values. Figure 6.2 illustrates the region indexing in the SPX framework for a 2D example.

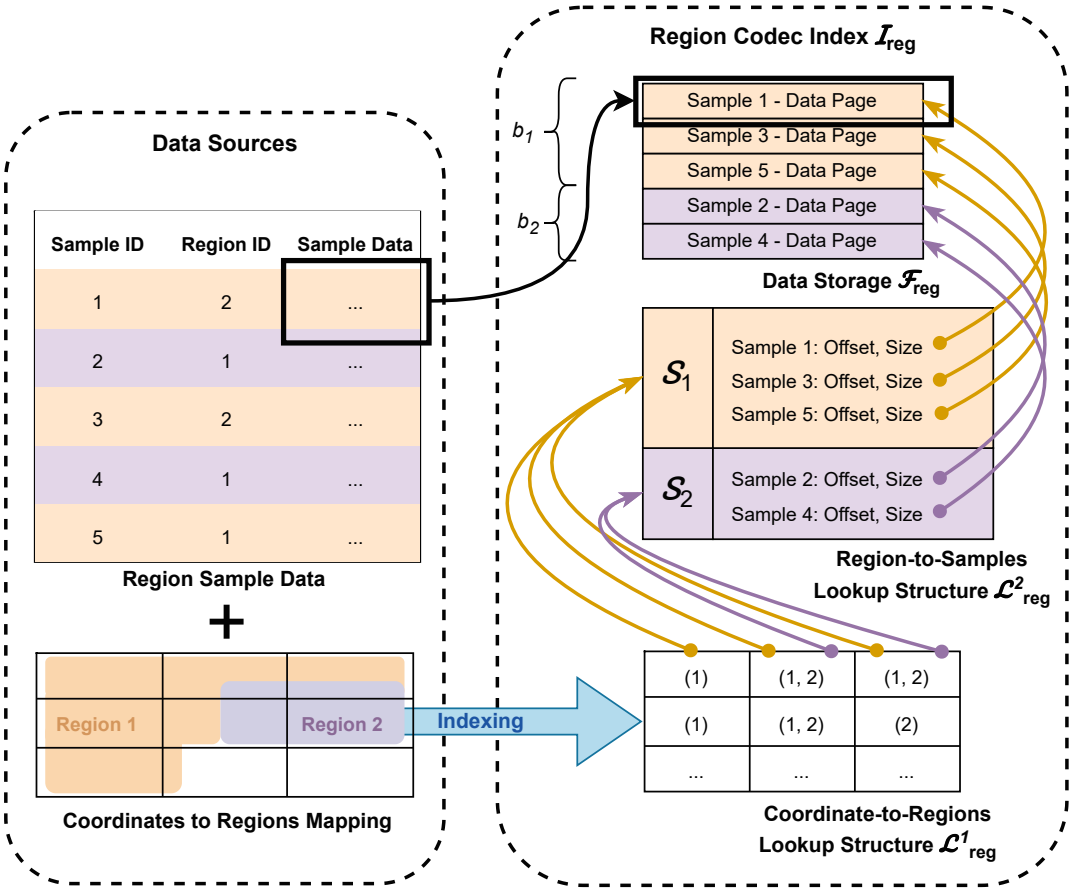

Figure 6.2: Region grid indexing in the SPX framework. $\mathcal{L}^1_{\text{reg}}$ is in this case a 2D structure containing the region ItemIDs for each coordinate. $\mathcal{L}^2_{\text{reg}}$ is a hashmap containing the region sample information for each region ItemID.

6.3 Creating Indices - The Indexing Process

A new spatial index is created by defining a reference space to index, selecting a space-filling curve to define a traversal path through the reference space, choosing a codec for the creation of the index data, and specifying the index items. Following the two types of indexing in SPX—voxel grid indexing and region indexing—the indexing of the reference space and the creation of the index differ. Figure 6.3 gives a high-level overview of the creation process of an index.

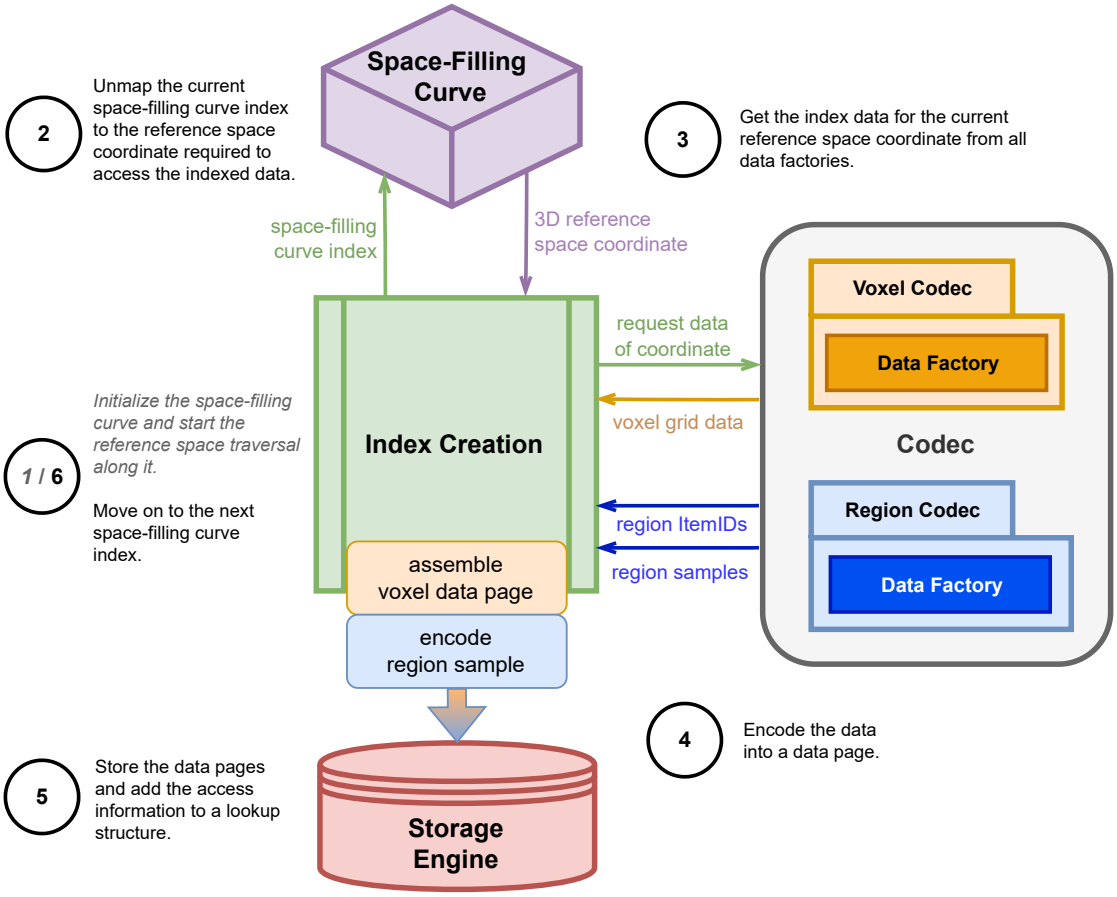

Figure 6.3: An overview of the index creation process in the SPX framework.

Creating Voxel Codec Indices - Indexing Voxel Grid Data

A voxel codec index indexes items containing voxel grid data. The codec is responsible for creating data entries for the index items at all coordinates. The codec does this by initializing a data factory for each index item, which must follow the data factory interface introduced in Section 5.3.1. The creation process of a voxel codec index is outlined in a simplified form in Algorithm 6.1.

Algorithm 6.1: Algorithmic description of the creation of a voxel codec index.

Input: *cName*: the voxel codec name
Input: *sfcName*: the space-filling curve name
Input: *storageEngineName*: the storage engine name
Input: *refDims*: the reference space dimensions
Input: *items*: the index items
Output: *voxelCodecIndex*: the created voxel codec index

```
1  vCodec ← getCodec(cName);
2  sfc ← initSpaceFillingCurve(sfcName, refDims);
3  se ← initStorageEngine(storageEngineName, refDims);
4  itemIDs ← [];
5  vFactories ← [];
6  foreach it in items do
7  |  itemIDs.append(createItemID(it));
8  |  vFactories.append(vCodec.createDataFactory(it));
9  end
   // traverse along space-filling curve
10 for idx ← 0 to sfc.getMaxIndex() do
11 |  x, y, z ← sfc.idxToPos3D(idx);
12 |  if x ≥ refDims.x or y ≥ refDims.y or z ≥ refDims.z then
13 |  |  continue;
14 |  end
15 |  for l ← 0 to dataLayers-1 do
   |  |  // assemble the data page for this coordinate
16 |  |  dataPage ← [];
17 |  |  for i ← 0 to length(itemIDs)-1 do
18 |  |  |  dataEntry ← vFactories[i].getItemDataEntry(x, y, z);
19 |  |  |  dataPage.append(itemIDs[i], dataEntry);
20 |  |  end
21 |  |  se.setVoxelData(l, x, y, z, idx, dataPage);
22 |  end
23 end
   // the storage engine is the container of the index
24 voxelCodecIndex ← se;
25 return voxelCodecIndex;
```

Creating Region Codec Indices

Similarly, for a region codec index, a coordinate space is traversed according to a space-filling curve, establishing the data order. For each collection of region data, the region codec creates a data factory. While in a voxel codec, the data factory of an item is responsible for creating the data of one index item at a coordinate, in a region codec, a region data collection's data factory is responsible for creating the data of all samples of a region. Algorithm 6.2 outlines the process for creating a region codec index.

Region data pages consist of the data pages for all samples within a region. However, the region data pages, and therefore the sample data pages, are written only once for each region, when it first appears during the reference space traversal. The sample data pages are stored separately, even though written as a consecutive block of data pages (the region data page), to allow for the efficient retrieval of the possibly large single sample data pages. Although a region is only a meta-structure to group the region samples, both the region and the region samples require an ItemID, either of type *region* or type *sample*, to be uniquely identified.

After creating indices, hence indexing the available data to optimize data access, the indices can be queried. The querying process is described in the following section.

Algorithm 6.2: The creation of a region codec index.

Input: *cName*: the region codec name
Input: *sfcName*: the space-filling curve name
Input: *storageEngineName*: the storage engine name
Input: *refDims*: the reference space dimensions
Input: *dataCollections*: the region data collection containing the region samples
Output: *regionCodecIndex*: the created region codec index

```
1  rCodec ← getCodec(cName);
2  sfc ← initSpaceFillingCurve(sfcName, refDims);
3  se ← initStorageEngine(storageEngineName, refDims);
4  rFactories ← [];
5  foreach it in items do
6  |  rFactories.append(rCodec.createDataFactory(it));
7  end
8  regionsSaved ← [];
   // traverse along space-filling curve
9  for idx ← 0 to sfc.getMaxIndex() do
10 |  x, y, z ← sfc.idxToPos3D(idx);
   |  // skip the coordinate if it is outside of the reference space
11 |  if x ≥ refDims.x or y ≥ refDims.y or z ≥ refDims.z then
12 |  |  continue;
13 |  end
14 |  regionIDsOfCoordinate ← [];
15 |  foreach rFactory in rFactories do
16 |  |  regionIDs ← rFactory.getRegionIDsAtCoordinate(x, y, z);
17 |  |  regionIDsOfCoordinate.append(regionIDs);
18 |  |  for l ← 0 to dataLayers-1 do
19 |  |  |  forall regionID in regionIDs do
20 |  |  |  |  if regionsSaved.contains(regionID) then
21 |  |  |  |  |  continue;
22 |  |  |  |  end
   |  |  |  |  // save all the region samples of the region
23 |  |  |  |  sampleIDs ← rCodec.getRegionSampleIDs(regionID);
24 |  |  |  |  forall sampleID in sampleIDs do
25 |  |  |  |  |  sample ← rCodec.getRegionSampleItem(sampleID);
26 |  |  |  |  |  encodedSample ← rCodec.encodeRegionSampleItem(sample);
27 |  |  |  |  |  se.setRegionSampleData(l, regionID, sampleID, encodedSample);
28 |  |  |  |  end
29 |  |  |  |  regionsSaved.append(regionID);
30 |  |  |  end
31 |  |  end
32 |  end
33 |  se.setRegionsAtCoordinate(x, y, z, idx, regionIDsOfCoordinate);
34 end
   // the storage engine is the container of the index
35 regionCodecIndex ← se;
36 return regionCodecIndex;
```

6.4 Querying Indices

The main goal of the SPX framework is to facilitate the observation and exploration of large volumes of volumetric image data, derivative data, segmented structures, and region samples based on aggregation queries within a time frame of a few seconds. To achieve this, several techniques are employed, integrating the various components discussed earlier. As outlined in Section 5.3, the codec determines the query types that can be executed on the index data by implementing the query processor interface. Initiating a query on an index involves specifying the query area, the query type, and the necessary query parameters. The query type must be one of the types supported by the codec of an index. Conceptually, a query can be divided into three parts: evaluating the query area to produce a set of reference space coordinates, retrieving data from the storage engine that manages the index data, and processing the data using an initialized query processor from the index's codec. A conceptual overview is provided in Figure 6.4, and the process in its basic form is described in Algorithm 6.3.

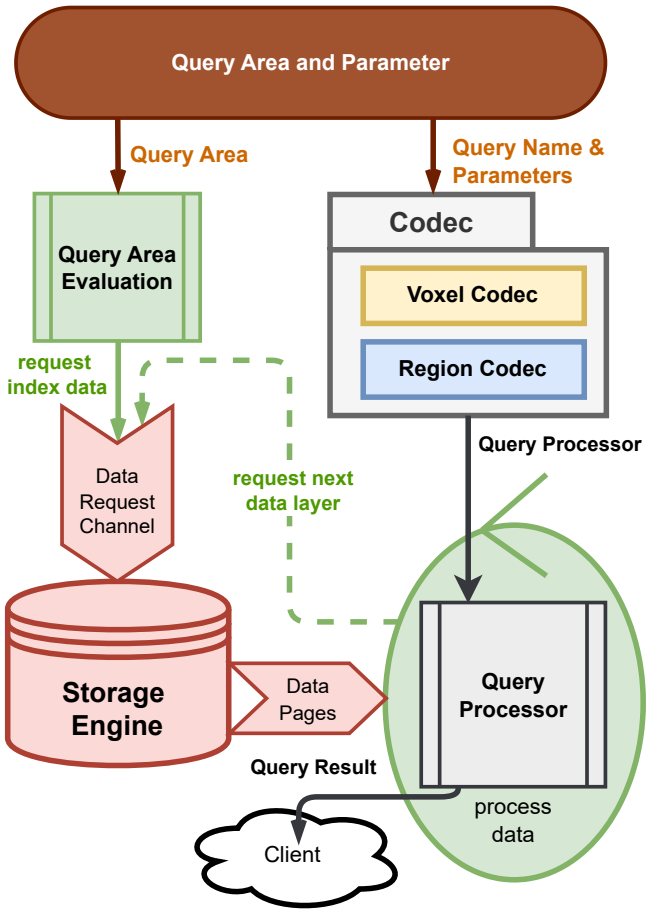

Figure 6.4: A high-level overview of the query processing in the SPX framework.

Algorithm 6.3: Index Queries - The basic query process outlined as simplified pseudo-code.

```
Input: codec: the codec of the index
Input: se: the storage engine of the index containing the data and data access methods
Input: sfc: the space-filling curve of the index
Input: queryCoordinates: the query area coordinates
Input: queryTypeName: the query type
Input: queryParams: the query parameters
1  qProcessor ← codec.initQueryProcessor(queryTypeName, queryParams);
   // get the space-filling curve indices of the query area coordinates and order
      them according to the space-filling curve
2  coordPairs ← [] foreach c⃗ ∈ queryCoordinates do
3  |  sfcIdx ← sfc.pos3DToIdx(c⃗);
4  |  coordPairs.append([sfcIdx, c⃗]);
5  end
6  sortedCoordPairs ← sortCoordPairsBySFCIdx(coordPairs);
7  if codec is voxel codec then
8  |  foreach coordPair in sortedCoordPairs do
9  |  |  dataPage ← se.getVoxelData(coordPair);
10 |  |  dataEntries ← [] for entryIdx ← 0 to dataPage.entryCount-1 do
11 |  |  |  itemID ← extractItemID(dataPage[entryIdx]);
12 |  |  |  dataEntry ← extractDataEntry(dataPage[entryIdx]);
13 |  |  |  dataEntries.append(dataEntry);
14 |  |  end
15 |  |  qProcessor.handleCoordinateResult(dataEntries);
16 |  end
17 end
18 if codec is region codec then
19 |  regionItemIDs ← [];
20 |  foreach coordPair in sortedCoordPairs do
21 |  |  regionItemIDs.extends(se.getRegions(coordPair));
22 |  end
23 |  uniqueRegionItemIDs ← createUniqueRegionItemIDs(regionItemIDs);
24 |  foreach regionID ← uniqueRegionItemIDs do
25 |  |  sampleIDs ← se.getRegionSamples(regionID);
26 |  |  foreach sampleID ← sampleIDs do
27 |  |  |  sampleData ← se.getRegionSampleData(regionID, sampleID);
28 |  |  |  sampleItem ← codec.decodeRegionSampleDataEntry(sampleData);
29 |  |  |  qProcessor.handleRegionSampleItem(sampleItem);
30 |  |  end
31 |  end
32 end
33 return qProcessor.getQueryResults();
```

Index Data Flow

One fundamental aspect of the SPX framework is its query engine, which follows a *producer-consumer pattern* [51] with a single producer, the storage engine, and multiple consumers. As outlined in Section 5.5, the storage engine utilizes Go's buffered channels. These channels act as safe conduits for passing data between concurrent Go-routines, eliminating the need for locks or synchronization primitives. Calling routines can read from or write to the channels without being blocked, as long as the channel's buffer is not full. This non-blocking operation is essential for maintaining the data pipeline's efficiency and throughput. The storage engine uses two channels: one for handling data fetch requests and another for providing the fetched data as responses. Due to the non-blocking nature of the channels, the storage engine can continue processing requests without waiting for the query engine to consume the data. This asynchronous operation is crucial for maintaining a steady *production* of data and preventing pipeline stalls. Figure 6.5 provides an overview of the storage engine's data flow during a query.

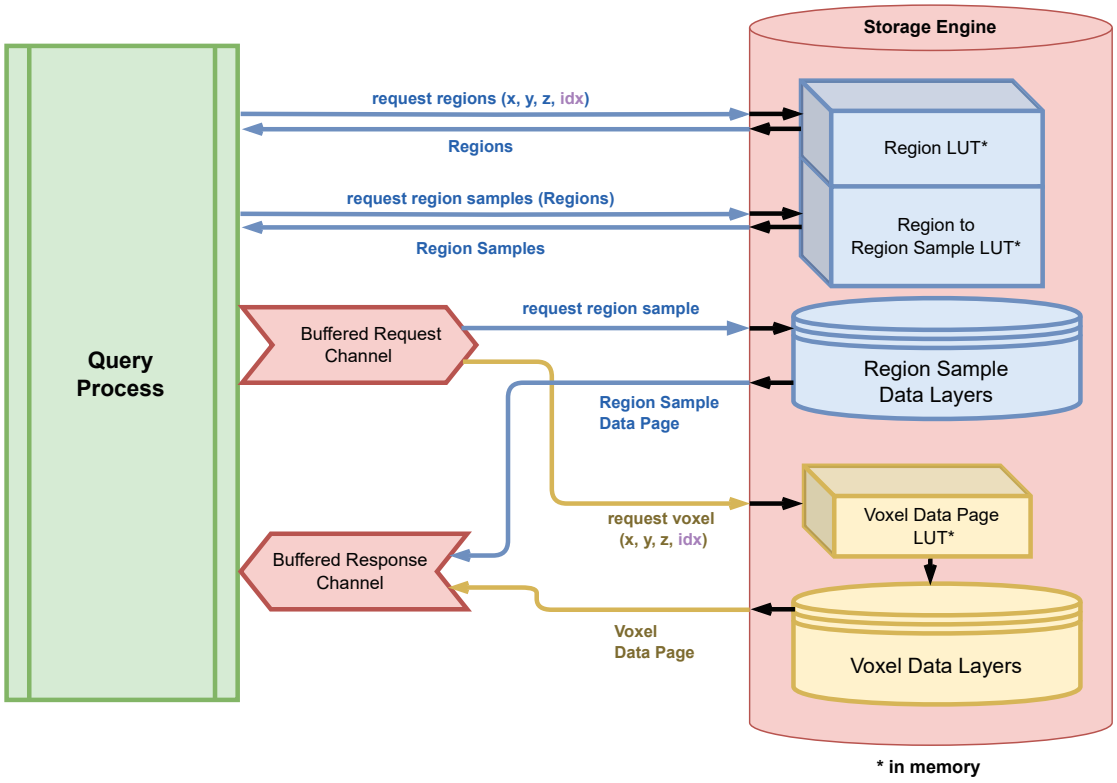

Figure 6.5: An overview of the storage engine, illustrating the data flow during a query in the SPX framework, including how Go's channels are used for asynchronous data-handling.

The storage engine is responsible for facilitating the access to the data pages of the index data. This is achieved using lookup structures such as a 3D array of required information,

as outlined in Section 8.1. In an index handling voxel grid data, this lookup structure is used internally by the storage engine to map coordinates to data pages. However, in an index handling region data, its lookup structures must be accessible from outside the storage engine, allowing the query engine to retrieve the region ItemIDs for the queried coordinates in the first mapping step. This coordinate-to-region lookup structure of the storage engine is typically similar to the one used for voxel grid data (i.e. a 3D array).

Mapping the query area to region ItemIDs often results in duplicates, as multiple coordinates in the query area may overlap with the same region. To address this, the query engine creates a unique set of region ItemIDs. This set is used to access region samples for all regions by utilizing the storage engine's second lookup structure. This structure is typically a 2D array of region ItemIDs and sample information. The sample information includes details such as the sample ItemID and the access information for the sample's data page, all ordered along the space-filling curve. Since each region and each region sample is indexed separately, the query processor can filter the regions and region samples using their ItemID information (e.g., the dataset). This filtering occurs before the query engine sends the fetch request for the sample data to the storage engine's request buffer. This selective approach helps to reduce the reading workload on the storage engine and to minimize the decoding and processing effort.

6.4.1 The Multi-Routine Query Engine

One major reason for choosing the Go programming language for the implementation of SPX is its built-in concurrency model, which is based on Go-routines and channels, as outlined by Rob Pike [67] and the Go memory model section of the documentation [81]. Go-routines are lightweight threads that the Go runtime schedules and distributes across available CPU cores, enabling concurrent and parallel execution of functions. Channels provide a lock-free mechanism for communication between Go-routines, allowing them to synchronize and exchange data efficiently. Channels can be buffered, enabling asynchronous data exchange, or they can be unbuffered, enabling synchronous communication. The combination of Go-routines and channels offers a powerful and efficient way to implement concurrent and parallel algorithms, making them ideal for the SPX framework's query engine.

The storage engine exposes its response channel, allowing the query engine to retrieve data pages from the queried index. It creates a processing job for each retrieved data page. A job is a function that decodes individual data entries—such as the index items at a specific coordinate in a voxel codec index, or a single region sample in a region codec index. Once the decoding is complete, the job invokes the query processor's data-handling function. If additional data layers are required, the job sends a new fetch request to the storage engine.

Worker routines handle the execution of jobs concurrently. Creating a new worker routine for each job would be inefficient, and can even be harmful for the stability of SPX if the number of workers is not limited. Therefore, SPX employs a routine pooling approach

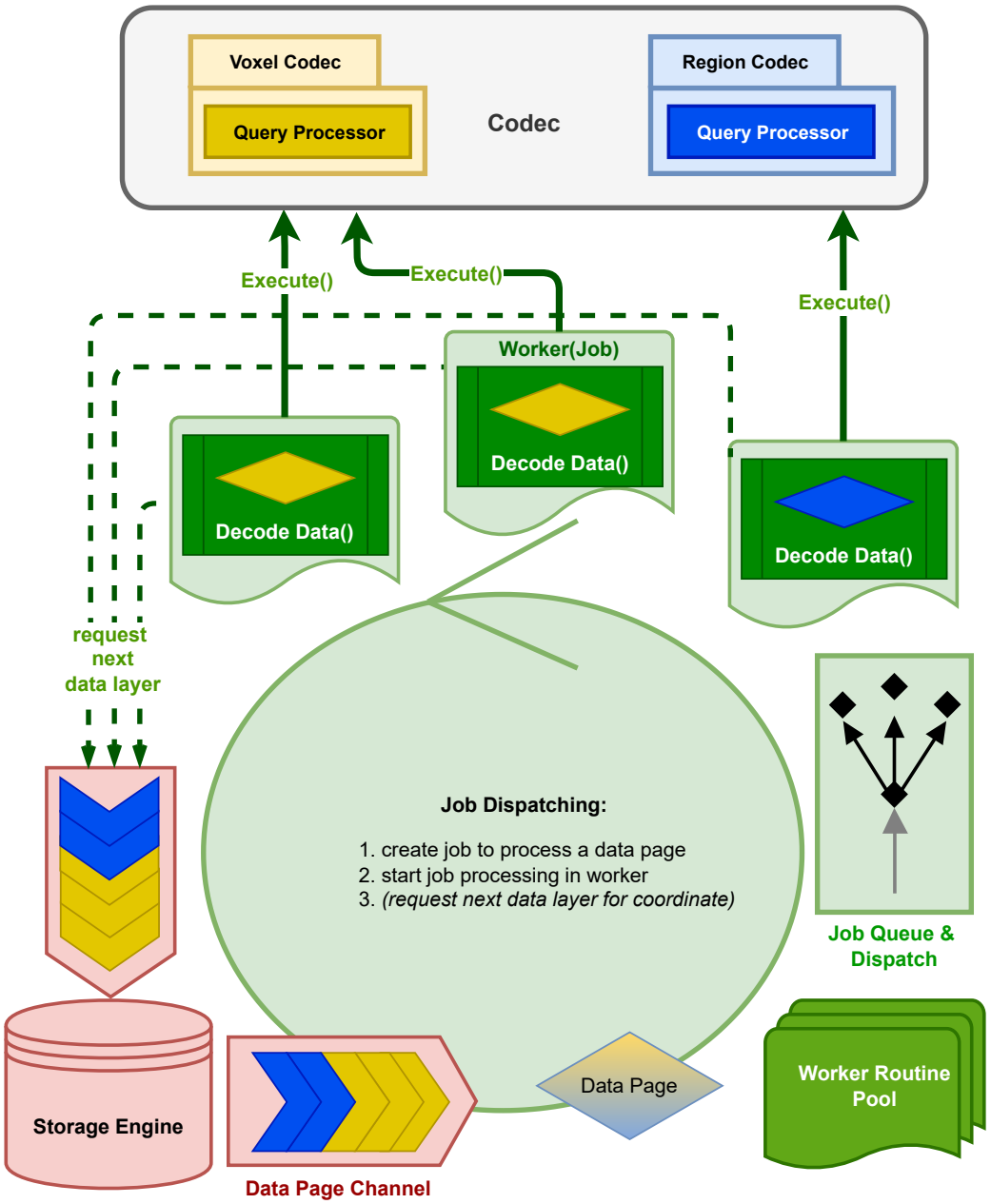

Figure 6.6: Overview of the Multi-Routine Query Engine in SPX. This diagram illustrates the workflow of the SPX query engine's multi-routine job-dispatch system. The data pages retrieved from the storage engine are processed concurrently by multiple worker routines. The system efficiently handles the execution of numerous processing jobs in parallel, leveraging Go's concurrency model to optimize query performance across multiple CPU cores. The query processor decides whether to fetch data for the next layer of the coordinate or region sample.

described in the next section. Jobs are dispatched to the routine pool via an unbuffered channel. Since the query engine only creates jobs and sends them to the routine pool via this unbuffered channel, it must wait for the job to be picked up, if all worker routines are busy. Figure 6.6 provides an overview of concurrent query data processing.

Routine Pooling

Creating and destroying a large number of routines can be resource-intensive. Although Go-routines are lightweight, there is still overhead in spinning up each one, including allocating its own stack (initially 2 KB). This becomes problematic in scenarios with many short-lived tasks, where excessive Go-routine creation can lead to significant memory and scheduling overhead. To mitigate this, SPX employs a Go-routine pool.

Upon initialization, a number of Go-routines are pre-allocated typically ten times the number of CPU cores available on the system. This number is based on performance measurements and can be adjusted as needed. These pre-allocated routines remain in a suspended state until the dispatcher activates them to process a job. The dispatcher is another Go-routine listening on the job channel. When a job is received, the dispatcher assigns it to an available worker routine. Instead of being destroyed after job completion, the routine returns to the pool and remains in a suspended state. This ensures continuous and efficient job execution. If no routine is available, the pool can spin up a new one, provided that the maximum routine limit has not been reached. This maximum limit can be set as a startup parameter or default to a predefined value, such as 100 times the number of CPU cores. The purpose of the routine pool is to keep the number of Go-routines, stack memory, and runtime scheduling overhead at a manageable level under normal conditions while allowing for scaling concurrency as needed if processing the index data becomes the performance bottleneck.

6.4.2 Processing the Index Data

The multi-routine job-dispatch system significantly increases the efficiency and speed of processing retrieved data pages. It helps prevent data bottlenecks and improves system scalability by efficiently allocating and managing available resources. However, this system demands more memory than a traditional blocking read-and-process system, as multiple data pages may reside in memory simultaneously. For indices with large data pages, this can lead to high memory consumption. To address this, the number of worker routines can be limited, and the buffer size for data pages in the storage engine's response channel can be reduced. This prevents the storage engine from fetching more data pages until the query engine has processed the current ones, thereby reducing the system's memory footprint. However, this approach may also decrease system throughput. The optimal balance between memory consumption and system throughput must be determined based on the system's requirements, the available resources, and the index data and query processing requirements.

As mentioned before, each query initializes a query processor specific to the index codec. The query processor implements the query logic and keeps track of the query state, allowing on-the-fly aggregation of query results. Multiple worker routines can execute the query processor's data-handling function concurrently, as seen in Figure 6.6, so the implementation of the query processor must be thread-safe. The data-handling function returns a boolean indicating whether to fetch data for the next layer of the coordinate or region sample. Once the query processor has determined that no further data from deeper layers is needed for all data pages in the query, it finalizes the query result aggregation. The query processor then returns the aggregated result to the query engine, which in turn returns it to the query caller.

Given the system's asynchronous architecture, it is impossible to guarantee that jobs will be completed in the same order they were created and dispatched. As a result, data fetch requests for subsequent layers may be created out of the initially optimized order. Depending on the storage engine and its storage medium, this can lead to degraded performance due to necessary jumps between file regions. To address this, SPX utilizes a forward-read-flow synchronization mechanism, outlined in the following section.

6.4.3 Forward-Read-Flow Synchronization

Initially, read requests are optimized to be read only forward direction along the space-filling curve, minimizing cache misses in block-based storage media. However, as query evaluation progresses to deeper layers, data fetch requests for the storage engine may be created out of the original optimized order due to the asynchronous nature of data page processing. To maintain the optimized order, the query engine uses a backlog structure to buffer out-of-order coordinates and sample requests. Like the storage engine and the routine pool, the forward-read-flow synchronization mechanism also uses an unbuffered channel to receive data fetch requests from multiple worker routines. Once a request arrives, the mechanism checks whether it is the next request to maintain the optimized order. If not, the request is stored in the backlog until either the expected request arrives, or the expected coordinate/sample is marked as complete, at which point the next request in the optimized order is processed, if it is stored in the backlog. This backlog queue helps avoid jumps between data areas in the storage medium, though its effectiveness depends on the specific storage engine and medium used such as block-based storage or in-memory storage. For example, in an in-memory storage engine that keeps all index data in the main memory, the backlog queue would only introduce unnecessary overhead. Therefore, the storage engine has a method for the query engine to determine whether the backlog structure should be used or not.

6.4.4 Bringing it all Together - The Multi-Routine Query Engine

Algorithm 6.4 describes how the query engine works, including the job-dispatch, the routine pool, and the forward-read-flow synchronization. This description uses the context of a voxel codec index, but the same principles apply to a region codec index. Instead of using the coordinates directly to obtain the data pages from the storage engine, in case of a region codec index, the sampleIDs after the two mappings (coordinates-to-regions, and regions-to-sample) are used to access the data pages of the region samples. As with the voxel codec, it depends on the storage engine and the storage medium whether the forward-read-flow synchronization is enabled or not. For simplicity, the current description of the multi-routine query engine assumes a single query. However, SPX's query engine supports the execution of multiple queries on the same index simultaneously, though this is subject to certain restrictions, as outlined in the following section.

6.4.5 Multi-Queries

SPX supports the parallel execution of multiple queries on the same index, provided that the queries operate within the same query area. For each query, a separate query processor is initialized. These query processors receive the queried index data in the sequence of their initialization. To manage this parallel processing, a bit-flag is associated with each coordinate or region sample. This bit-flag tracks which query processors still require data from the current layer. Each bit within the flag corresponds to a specific query processor. When a bit is set, it indicates that the associated query processor no longer needs data from that coordinate or region sample for its result aggregation. Once all query processors have signaled that they have finished processing the data for a particular coordinate or region sample, that data is discarded. This process continues until all coordinates have been processed, ensuring that all query processors have completed their result aggregation. The final results are then returned in the order of the processors' initialization. This approach allows for efficient multi-query execution on the same index with the same query area, eliminating the need to run multiple queries sequentially.

Algorithm 6.4: Query Engine - pseudo-code of the query engine's doings while working on a query on a voxel codec index. (simplified)

Input: *se*: the storage engine of the index
Input: *sortedCoordPairs*: the coordinate pairs of the $\vec{v}$ and the space-filling curve index
Input: *queryProcessor*: the initialized query processor of the index's voxel codec
Output: *queryResults*: the aggregated query results

```
1  dataRequestChn ← se.getRequestChannel();
2  dataResponseChn ← se.getResponseChannel();
3  readFlowSync ← initReadFlowSync();
   // this channel receives a signal once the query is done
4  queryDoneChn ← createUnbufferedChannel<bool>();
5  foreach coordPair in sortedCoordPairs do
6  |  dataRequestChn.send(coordPair);
7  end
8  jobChn ← createUnbufferedChannel<Job>(); // the job receiving channel
9  go routine: Function JobDispatch():
10 |  while voxelDataPage ← dataResponseChn.get() do
   |  |  // create a job to process the data and dispatch it to the routine pool
11 |  |  job ← Function JobLambda():
12 |  |  |  coordinatePair ← voxelDataPage.coordinatePair;
13 |  |  |  dataEntries ← extractDataEntriesOfPage(voxelDataPage.data);
14 |  |  |  needNextLayer ← queryProcessor.handleCoordinateResult(dataEntries);
15 |  |  |  if needNextLayer then
16 |  |  |  |  readFlowSync.addRequest(coordinatePair);
17 |  |  |  else
18 |  |  |  |  readFlowSync.finishCoordinate(coordinatePair);
19 |  |  |  end
   |  |  |  // if the next expected request is in the backlog, send it
20 |  |  |  dataRequest ← readFlowSync.getNextRequest();
21 |  |  |  if dataRequest then dataRequestChn.send(dataRequest) ;
22 |  |  |  else if !readFlowSync.hasOutstanding() then
23 |  |  |  |  queryDoneChn.send(true); // all coordinates processed
24 |  |  |  end
25 |  |  end
26 |  |  jobChn.send(job);
27 |  end
28 end
   // this happens inside the routine pool
29 go routine: Function RoutinePool():
30 |  while job ← jobChn.get() do
31 |  |  workerRoutine ← RoutinePool.getWorkerRoutine();
32 |  |  workerRoutine.execute(job);
33 |  |  RoutinePool.returnWorkerRoutine(workerRoutine);
34 |  end
35 end
36 queryDoneChn.get(); // wait until the query is done
37 return queryProcessor.getQueryResults();
```

CHAPTER 7

Implementation Details

This chapter provides insights into specific implementation details of the SPX framework. The initial implementation of the distance-field index was developed by Bruckner et al. [12] in Matlab, while Schulze et al. [76] proposed and implemented the staining index in Java. Subsequently, Schulze began porting the Java implementation to Go, leading to the development of the initial prototype mentioned in Section 6.1. The current implementation of SPX builds upon Schulze's Go version. Go is a modern programming language that is statically typed, compiled, and garbage-collected, supporting multiple paradigms. The decision to continue using Go was made due to its advantageous properties for asynchronous multi-routine data processing, its comprehensive standard library, and its strong performance characteristics.

In this chapter, I detail the implementation specifics of the SPX framework, focusing on aspects that, while important, do not directly contribute to the overall conceptual understanding of the framework. The chapter is organized as follows: First, I explain the *header*, which serves as a descriptor of the general properties of an index. Next, I provide details on the query areas and the internal usage of the ItemID, as outlined in Section 5.1. Following that, I describe the creation and management of data pages containing the index data, which are stored by the selected storage engine. This chapter also discusses the currently available storage engine implementations, which are based on different concepts, along with implementation details for the currently available space-filling curves. The chapter concludes with a brief description of the merging of indices and the process of creating indices with many index items.

7.1 The Header - General Index Properties

Each index provides essential information to determine its suitability for queries. This information is stored in the header and is used by the index tools to identify the appropriate index. The header contains the following properties:

1. The **reference space name** identifying the reference space for the indexing.
2. The **reference space dimensions**.
3. The name of the used **codec**.
4. The number of voxel data **layers** and region data layers.

Queries are always executed on a per-reference space basis, making the reference space the primary indicator of whether an index is suitable for handling a query. The codec is necessary to understand the type of data the index contains and to determine which queries the index can process. Besides the header, also the list of ItemIDs belongs to the index's properties, which might be of interest regarding running a query on an index.

7.2 ItemID List, ItemID Kinds, and Mapped ItemIDs

As previously outlined in Section 5.1, each index item is associated with a unique identifier consisting of *dataset key*, *type*, and *item key*. In all communication with clients, the ItemIDs are serialized as strings of the form *dataset:type:item*. Internally, the type is represented as a one-byte enum value, therefore the *type*-string in the serialized ItemID must be of a known value. As with other major parts of the SPX framework, the ItemID is also defined by an interface and new ItemID kinds can be added by implementing this interface.

The SPX framework currently supports two kinds of ItemIDs, one where the dataset key and the item key are integers, and one where they are strings. The kind of ItemID used is typically determined by the index's usage environment. For instance, entries in relational databases typically use integer primary keys. Consequently, datasets and items have integers as keys. If the serialized ItemID conforms to this pattern, with both the dataset key and item key as integers, the index will use fixed-size integer ItemIDs. However, some databases, such as most NoSQL systems, primarily use unique string keys. To cover environments where this is the case, the second ItemID implementation was created, with both the dataset key and item key as strings. SPX automatically selects the appropriate ItemID type based on the serialized ItemIDs provided during index creation.

Having complex ItemIDs, especially of arbitrary length strings, is problematic. In a voxel codec index, this identifier is written every time its corresponding index item has a data entry at a coordinate, exponentially growing the storage requirements of the index, as well as the memory requirements during the execution of a query. For a region index,

the ItemIDs are kept multiple times in memory within the lookup structures, and large ItemIDs raise the memory footprint of the index. To address this issue, a simple mapping solution has been implemented.

When an index structure is built, every added ItemID is added to the list of ItemIDs. Instead of writing the ItemID into a data page in case of a voxel codec index, or into the region and sample lookup structures of a region codec index, the index internally uses only the list index of the ItemID. This list index of an ItemID, called *mapped ItemID*, is used throughout the entire data handling and querying process of an index instead of the ItemID. While this approach reduces the memory pressure drastically, the disadvantage is the need for an additional unmapping during the query result serialization, but since the list index is used, this is very efficient.

If queries require an ItemID as a parameter, the corresponding mapped ItemID must be found. In most cases, a simple list search is enough, since most queries only require one ItemID as parameter. For cases where an index has many ItemIDs, and takes many ItemIDs as query parameters for its codec's queries, SPX can create a reverse mapping using the serialized ItemIDs as key of a dictionary, and the mapped ItemID as value. The ItemID array, as well as the optionally created dictionary, reside in memory while the index is loaded, and is restored by the storage engine upon opening of an index. In case of region codec indices, the data access can also be facilitated by using the region ItemIDs. Typically multiple regions are requested, and in that case, the reverse mapping of the ItemID provides significant performance improvements. The data access in case of voxel codec indices can be only facilitated using query areas, to which more details are given in the next section.

7.3 About Query Areas

The query area is sent from the requesting client and is evaluated in SPX's query engine. A query area can consist of several *brushes* and *masks*, which all get converted to 3D coordinates. An overview of the query area evaluation can be found in Figure 7.1.

A *brush* is a list of coordinates with a scalar value as the radius defining a sphere around each coordinate. Any coordinate within a sphere is included in the query. The coordinates of a brush are typically close to each other, resulting in significant overlap between the spheres defined by each brush coordinate. Therefore, coordinate deduplication is necessary. Instead of calculating all the coordinates within each sphere, a bounding box is created around the brush coordinates, and each coordinate within the bounding box is checked to see if it lies within the sphere of any given brush coordinate. While a brush defines spheres around coordinates, a *mask* provides a memory-efficient way to represent a set of coordinates.

A *mask* in SPX is a set of coordinates encoded in a memory-efficient format. Transferring a large number of coordinates in vector format is an expensive task due to the sheer volume of data involved. A mask consists of a bounding box, defined by one corner

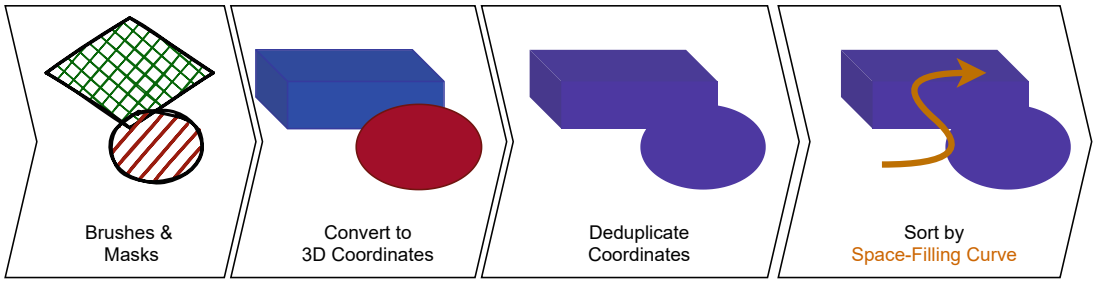

Figure 7.1: Evaluation of a query area to the respective index coordinates.

coordinate and its dimensions. Each coordinate inside the bounding box is represented by a bit in a character array. If the bit is set, the coordinate is part of the query coordinates.

After each brush and mask has been evaluated, the bounding boxes of the brushes and masks are checked for overlap, and the coordinates within the overlapping bounding box areas are deduplicated. The deduplication happens similarly to the mask encoding, by bit-masking coordinates in a bounding box spanning all overlapping areas. The final set of coordinates is subsequently used to query the index, and therefore access the data pages containing the index data.

7.4 Data Page Creation and Handling

This section provides implementation details on the creation and handling of the data pages. As mentioned in the Key Terms Section 4, a data page is a contiguous block of memory managed by the storage engine. It contains the index data for a specific coordinate in case of voxel grid data, or a specific region sample in case of region data.

In a voxel codec index, the data page holds all the collected tuples of mapped ItemIDs and their corresponding data entries at a particular coordinate. These tuples are concatenated into a single data page for that coordinate. Upon execution of a query, the data pages are retrieved from the storage engine, decomposed into individual tuples of mapped ItemIDs and data entries, and processed by the query processor. In the context of a region codec, a data page contains the encoded data of a region sample. The encoding and decoding of this data depends on the complexity of the region sample and are specific to the region codec being used. Since a data page is simply a contiguous block of memory, it can optionally be compressed before it is saved in the index. Given the potential size of data pages, particularly in large indices, managing storage efficiently becomes crucial. This brings us to the concept of Inline Data Page Compression.

Inline Data Page Compression

With an increasing number of index items, the data pages in a voxel codec index also increase in size. Similarly, the more information stored in a region sample, the larger the sample's data page becomes. One method to reduce the storage memory used by an index is to apply compression. Since the data pages are stored and read separately, compression is applied on a per-data page level. Data pages are compressed using one of the available compression mechanisms selected by the index-creating user. By leveraging the *Reader* and *Writer* interfaces in Go, it's straightforward to incorporate compression methods from publicly available external libraries. Currently the following compression mechanisms are integrated: *Deflate*, *Zstd*, *Snappy/Snappy2*, and *LZ4*.

However, in many cases applying data page compression is not beneficial. If the buffer being compressed is small or contains data that is not well compressible, compression algorithms can increase the overall content size. To mitigate this issue, a practical solution is to compare the compressed and uncompressed data page sizes and use the compressed version only if the compression ratio exceeds a reasonable threshold. To indicate whether a buffer is compressed or uncompressed, a one-byte flag is written at the beginning of a data page when it is stored. Since inline compression reduces the required storage memory, it also enables indices that reside purely in memory to contain more index items. The use of Go's standard *Reader* interface allows the compression to be handled transparently, without influencing the deserialization of the data.

While compression reduces the amount of data to be stored in or read from the storage engine, it also requires additional CPU and memory resources to decompress the compressed data page. This can prolong the query handling process, except in cases where the main bottleneck are the I/O operations of the storage engine. In some compression mechanisms, like LZ4, multithreading is possible, but it is counterproductive in SPX's use case with its parallel data page processing, described in Section 6.6.

In a coordinate's data page of a voxel codec index, the individual entry tuples of mapped ItemIDs and data entries need to be unpacked. Without compression, this can be done directly by creating a *view* on the byte array of the page. A *view* provides restricted and limited access to an underlying byte array without creating a copy of the data. Each data entry for an index item at a coordinate is represented as a *view* on the data page's byte array. When compression is enabled on a data page, it needs to be uncompressed into a temporary buffer first, before the *views* of the data entries are created. To avoid constantly allocating and freeing temporary buffers of approximately the same size, a buffer pooling mechanism is used to keep the temporary buffers allocated. The buffer pooling mechanism is described in the following section.

Buffer Pooling

Buffer pooling is commonly used in systems with high rates of data input and output. Instead of repeatedly allocating and deallocating temporary memory for small, short-lived tasks, buffer pooling maintains a pool of pre-allocated memory blocks. When a new buffer is needed, one is obtained from the pool, and when it is no longer needed, it is returned to the pool instead of being deallocated by the garbage collector. This can reduce the overhead associated with memory allocation and deallocation, lower overall memory usage by reusing allocated buffers, and improve performance by reducing the frequency of expensive system calls, thereby reducing the strain on Go's garbage collection.

The buffer pool is an optimization utilized during query executions. If a query requires the allocation of a buffer, the buffer should be obtained from the pool. Once the buffer is no longer needed, it should be returned to the pool. The pool is thread-safe and supports concurrent access by multiple Go-routines. It is used in the data page handling, as well as in the codec implementations, where temporary buffers are needed for the codec's operations, such as handling the compression of a data page, or a query processor's result aggregation. Usually, during a query, many of the required buffers are of similar sizes. Therefore, the implementation of the buffer pool only grows buffers as required, and keeps its storage allocated once it is returned to the pool. The buffer pool is implemented as a stack, where the buffers are pushed and popped. Once a query finishes, the buffer pool is emptied, and the allocated memory is released.

7.5 The Space-Filling Curves

The space-filling curve used in an index determines the order of its data pages. New implementations of space-filling curves can be integrated by adhering to the space-filling curve interface, as outlined in Section 5.2. Given that many space-filling curves necessitate their full precalculation before mappings and reverse mappings can be performed, an initialization method is indispensable. The interface comprises methods for the 3D-to-1D mapping and the reverse, along with the `getMaxIndex` method. This last method is essential during the index creation phase, as it determines the extent of distance required to traverse the entire reference space along the space-filling curve index axis. Most space-filling curves work best with dimensions that are powers of two, including the Z-order curve [58], which is often not the case for a reference space. When moving along the space-filling curve index axis, the mapping must be reversed to ignore 3D coordinates outside of the reference space. Currently, SPX offers two space-filling curves: a linear mapping and the Z-order curve. Implementation details of the linear mapping and the Z-order curve are provided in the following sections.

7.5.1 The Linear Mapping

A linear mapping of the 3D coordinate space onto a 1D coordinate axis is implemented, which is also a simple space-filling curve, and is defined as follows:

If we consider a 3D grid of size $N_x \times N_y \times N_z$, a point in this grid is identified by its coordinates (x, y, z), where $0 \leq x < N_x$, $0 \leq y < N_y$, and $0 \leq z < N_z$. The mapping of these 3D coordinates to a 1D coordinate u can be defined as follows:

$$u = x + y \cdot N_x + z \cdot N_x \cdot N_y \tag{7.1}$$

The linear mapping is not suitable for large indices, as it does not optimize the data locality, which is crucial for efficient data access. However, it is useful for testing purposes, to compare its performance with other space-filling curves, or for use cases where data locality is less critical, such as having the index data entirely in memory. Its advantage is the simplicity of the mapping, the ease of implementation, and the high computational performance of the mapping. Since data locality is typically important in the use cases SPX covers, the Z-order curve is the primary space-filling curve used in SPX.

7.5.2 Z-Order Curve Mapping

SPX uses a Z-order curve, explained in Section 3.1.3, as its default space-filling curve. It provides good locality preservation, high performance mapping capabilities, and is well-defined for 3D space mapping [58]. The implementation of the Z-order curve can be heavily optimized by using bitwise operations and chunk precalculations to speed up the interleaving process and reduce the computational cost. This is described in the following sections.

3D to Z-Order Curve Index

In the case of mapping, the default Z-order mapping implementation interleaves the bits of each coordinate value. This process is performed separately for each coordinate part, setting the corresponding bit in the Z-order indexing value. Baert [7] explained how this process can be optimized by precalculating byte-sized chunks and storing them in a lookup table. Bit-masking is used to retrieve the lookup table index for each byte chunk, which is then used to calculate the Z-order coordinate. Listing 11 shows this process as Go-code.

```go
// precalculate the interleaved 3D bits of all possibilities of blocks of 8
↪  bits
splice3 = make([]int, 256)
for i := 0; i < 256; i++ {
   r := 0
   for j := uint(0); j < 8; j++ {
      r = r | ((i>>j)&1)<<(3*j)
   }
   splice3[i] = r
}

func pos3dtozorderLut(px, py, pz uint16) uint64 {
   // retrieve the interleaved x/y/z parts by using the lookup table for the
   ↪  8-bit chunks
   x := uint64((splice3[int(px>>8)&0xff] << 24) | splice3[int(px)&0xff])
   y := uint64((splice3[int(py>>8)&0xff] << 24) | splice3[int(py)&0xff])
   z := uint64((splice3[int(pz>>8)&0xff] << 24) | splice3[int(pz)&0xff])
   // shift y an z into place, and merge the interleaved x/y/z parts to the
   ↪  Z-order index
   return x | (y << 1) | (z << 2)
}
```

Listing 11: In this optimized version, splice3 is a lookup table that stores precomputed interleaved 3D bits for all possible 8-bit blocks. The function pos3dtozorderLut retrieves these precomputed values to efficiently map 3D coordinates to a Z-order index. This optimization was done according to the explanations of Baert [7]. - *in Go*

Z-Order Curve Index to 3D

Unmapping, which involves reversing the interleaving of bits, works by masking the bits of the Z-order curve coordinate and setting them in the result coordinate axis. This can be also optimized by precalculating a lookup table of the non-interleaved bits of all three coordinates as seen in Listing 12. The unmapping is only required during the indexings, and therefore it is not performance-critical. The space-filling curve determines the order in which data pages are sent to the storage engine during the indexing process, and the unmapping is used to retrieve the reference space coordinate from the space-filling curve index during the reference space traversals along the space-filling curve. Since this Z-order implementation assumes cuboids with side lengths that are powers of two, some coordinates typically are outside the actual reference space. Therefore, the unmapped coordinates are checked against the reference space dimensions, and only valid coordinates inside the reference space are used to retrieve the data entries, create the data pages, and store them in the storage engine. SPX supports several storage engines, each optimized for different use cases or data access patterns, as described in the following section. It depends on the implementation of the storage engine, and the underlying data storage facility, if the data order is important, and if the space-filling curve index is used for the data access. The available storage engines are described in the following section.

```
// precalculate the un-interleaved 3D bits of all possibilities for 9-bit
↪  blocks and store them in unsplice3
// x, y, z are uint16, so 48 bits are required for the Z-order index, using
↪  6 chunks of 9 bits each
unsplice3 = make([][3]byte, 512)
for i := 0; i < 512; i++ {
   // x coordinate part... 0th, 3rd, 6th bit
   xp := (i & 1) | (((i >> 3) & 1) << 1) | (((i >> 6) & 1) << 2)
   j := i >> 1
   // y coordinate part... 1st, 4th, 7th bit
   yp := (j & 1) | (((j >> 3) & 1) << 1) | (((j >> 6) & 1) << 2)
   k := i >> 2
   // z coordinate part... 2nd, 5th, 8th bit
   zp := (k & 1) | (((k >> 3) & 1) << 1) | (((k >> 6) & 1) << 2)

   unsplice3[i] = [3]byte{ byte(xp), byte(yp), byte(zp) }
}


func zordertopos3dLut(zorder uint64) (uint16, uint16, uint16) {

   // look up the 9 bit chunk indices, retrieve the un-interlaced x/y/z
   ↪  parts
   p1 := unsplice3[int(zorder & 0x1ff)]
   p2 := unsplice3[int((zorder >> 9) & 0x1ff)]
   p3 := unsplice3[int((zorder >> 18) & 0x1ff)]
   p4 := unsplice3[int((zorder >> 27) & 0x1ff)]
   p5 := unsplice3[int((zorder >> 36) & 0x1ff)]
   p6 := unsplice3[int((zorder >> 45) & 0x1ff)]

   // combine the byte chunks by OR-ing and bit-shifting to the correct
   ↪  chunk bit positions
   x = uint16(p1[0]) | (uint16(p2[0]) << 3) | (uint16(p3[0]) << 6) |
   ↪  (uint16(p4[0]) << 9) | (uint16(p5[0]) << 12) | (uint16(p6[0]) << 15)
   y = uint16(p1[1]) | (uint16(p2[1]) << 3) | (uint16(p3[1]) << 6) |
   ↪  (uint16(p4[1]) << 9) | (uint16(p5[1]) << 12) | (uint16(p6[1]) << 15)
   z = uint16(p1[2]) | (uint16(p2[2]) << 3) | (uint16(p3[2]) << 6) |
   ↪  (uint16(p4[2]) << 9) | (uint16(p5[2]) << 12) | (uint16(p6[2]) << 15)

   return x, y, z
}
```

Listing 12: The precalculated LUT-based optimization of the Z-order unmapping, following the explanations of Baert [7]. The Z-order coordinate is split into chunks of 9 bits. All possible 512 coordinate values are de-interleaved into the 3-bit x/y/z parts of the unmapped coordinates and stored in a lookup table. To retrieve the unmapped 3D coordinates from a Z-order 1D coordinate, one needs to get the lookup table indices for the chunks by bitwise OR-ing and bit-shifting the chunk's bits. Once the 3D coordinate parts of the chunk have been retrieved, the chunks need to be fused by bit-shifting the chunk values to the correct bit index and OR-ing them into the final coordinate value. - *in Go*

7.6 Available Storage Engines

As described in Section 5.5, the storage engine manages data storage and implements the necessary methods of the storage engine interface. Several distinct implementations of the storage engine interface, each with unique attributes, are currently available. A storage engine's distinguishing features revolve around the storage medium it employs and the lookup structures and methodologies it utilizes to save and access data pages.

Besides the data pages containing the index data, the storage engine also stores the header, the list of ItemIDs, and the lookup structure it requires to access the data pages. All those parts are serialized and stored in the storage medium and are restored and kept in memory when the index is opened. The lookup structure in the storage engine depends on the codec type used in the index. For a voxel codec, the lookup maps either the 3D coordinates or the 1D space-filling curve indices, to the corresponding data pages. For a region codec, the storage engine provides also the intermediate mapping from coordinates to regions, and further from regions to region samples. In the last step, the region samples are mapped to the data pages containing the region sample data.

Although the current selection of storage engines all leverage local storage mediums, it's entirely possible to develop a storage engine that uses distributed or networked storage mediums, such as a remote database. This concept could serve as a cornerstone for future explorations in expanding the capabilities of the SPX framework. In the following section, three conceptually different storage engines are described: a key-value data store, a custom file-based storage engine, and an in-memory storage engine for small indices, described in the next section.

7.6.1 In-Memory Indices - The JSON/GOB Storage Engine

In-memory indices refer to indices that are compact enough to be stored in RAM, eliminating the need for costly I/O operations, and therefore improving access speeds. The only instances of hard drive I/O operations occur during the serialization of the storage engine's data structure to the storage medium, and the deserialization of the stored data structure back into memory upon opening an index. In-memory indices do not impose any restrictions regarding the implementation of the storage data structure.

An example implementation adopting maps for data storage and indexing is available in SPX. Several maps are used, each map storing data specific to the codec type used for an index. In the case of voxel codecs, the space-filling curve indices serve as the keys to access the data pages stored in a map. Each data layer is stored in a separate map for compartmentalization. For region codecs, three maps are employed: one to map space-filling curve indices to a list of regionIDs, a second to map regionIDs to the corresponding region samples, and a third to map the region samples to their data pages, with one data page for each data layer and region sample. This construct enables easy access to single region sample data on all layers.

The JSON/GOB storage engine harnesses Go's built-in serialization capabilities, enabling it to store data in either JSON or using Go's native binary serialization format GOB. During development, the JSON storage engine was extensively utilized due to its effortless and seamless access to the byte arrays that were stored. JSON also made it possible to inspect the serialized index using a regular text editor. To enhance the serialization performance, the storage engine was updated to optionally use GOB encoding. Although this introduced efficiency in data storage and loading, it stores data in a non-human-readable format, a trade-off for the performance benefits. In addition to memory constraints limiting the index size, the memory requirements for encoding and decoding the index onto the storage medium must also be considered. To reduce memory usage, the index data is serialized and deserialized in chunks. In-memory indices are not the typical use case of SPX, as the indices are expected to be large and not fit into memory. Another solution, which was simple to implement, was the usage of a key-value store.

7.6.2 A Key-Value Store - the BBolt Engine

A key-value store is a type of database that stores data as a collection of key-value pairs, where each value is associated with a unique key. This simple data model allows for fast and efficient retrieval of data based on its key. Key-value stores are often used in high-performance applications where speed and scalability are critical or as storage engines in higher-level databases like relational databases, and many implementations are available. Another advantage of key-value stores is their simplicity. In most cases, including in the SPX spatial indexing framework, key-value stores store save and recall encoded byte arrays (the data pages).

Although various solutions exist, the BBolt storage engine was chosen for the SPX framework. BBolt [22] is an embedded, high-performance key-value store written in Go. It utilizes a copy-on-write B+tree [91], a self-balancing, n-ary tree structure, which is stored in a simple memory-mapped file. The keys and values are stored in the tree's leaves, which also contain links to neighboring leaves for faster sequential access. BBolt seemed to be the most promising implementation since, according to the documentation, the usage of the B+Tree and the memory-mapping of one file leads to improved read performance but worse write performance compared to other key-value stores. It is easy to use, has no dependencies, and a well-defined API. BBolt also includes the concept of buckets, a mechanism to include separate key-value stores inside the same database file. In the proposed use case, BBolt's buckets are used to create a compartmentalization of different index parts. The following buckets are used in the current implementation:

- The **header bucket** stores the encoded header.
- The **ItemID list bucket** stores the index's list of ItemIDs.
- Each **voxel data layer** has a separate bucket. The space-filling curve's index is used as the key to access the data pages.

- A **region lookup bucket** for the region codec's mapping of the space-filling curve's index to a list of region ItemIDs.
- A **region sample lookup bucket** to access the list of region samples for each region.
- Each **region data layer** resides in its own bucket, and the region sample's mapped ItemID is used as the key to access the data page.

In BBolt, also the keys are byte arrays and are used to insert the data according to the key's byte order, defining the data order inside the B+tree. Adding new entries to the tree requires the database file to grow and the tree to be rebalanced. The growth always occurs in a certain size to avoid constantly regrowing the database file. BBolt offers a defragmentation mechanism to free acquired storage memory once the database is filled.

Despite its promising features, the development and testing of BBolt revealed certain drawbacks. Inserting many keys was found to be a time-consuming process due to the need for the tree to rebalance itself. To address this issue, I added a write-ahead-log structure on top of BBolt, and entries were written in batches in a single transaction. However, this approach did not result in significant improvements. Additionally, BBolt's database file grows by reserving memory in the leaves, resulting in a larger file with unused space. In BBolt, once a bucket is accessed, parts of its content is memory-mapped.

7.6.3 About Memory Mapping

Memory mapping is a concept used in BBolt, and also in the index's default storage engine solution, the memory-mapped index file described in the next section. According to Wolf [90], memory-mapping is a technique used by the operating system to translate parts of the physical memory of a file into virtual memory in RAM. When a file is mapped into memory, the operating system creates a mapping between the file's contents and a range of virtual memory addresses in the program's memory space. This allows the program to access the file's contents as if it were completely in memory. The mapping and loading of file data on demand happens automatically and transparently by the operating system to the executing process and is optimized using techniques such as on-demand paging, pre-fetching, read-ahead paging, and buffered writes. It enables the operating system to do this transparently and reduces the amount of costly I/O operations. This results in faster access times to areas in the file if they are already loaded into the main memory, especially if the program needs to access multiple areas of the file repeatedly. If the program tries to access a virtual memory address not currently loaded by the mapping, a page fault exception is thrown, prompting the exception handler to load the missing location into the memory-mapping and, consequently, into the main memory. This technique simplifies memory management and allows programs to interact with virtual addresses instead of raw physical memory, making the process more efficient and manageable. Memory mapping is crucial for the performance of our file-based storage engine, which is discussed in the next section.

7.6.4 The Memory-Mapped Index File

The memory-mapped index file is a solution similar to the one suggested by Schulze et al. [76], but extended to utilize memory-mapping. It is a densely written, single memory-mapped file with the following structure, where each part resides in a separate partition of the file. Given its scalability, memory efficiency, and good data access performance, the memory-mapped index file is the default storage engine for SPX. The file is divided into the following parts:

1. the index **header**
2. the **list of ItemIDs** of the index
3. the serialized **voxel lookup structure** for the voxel codec
4. the **voxel data layers** of the voxel codec, each in a separate partition
5. the serialized **region lookup structures** for the region codec
6. the **region data layers** of the region codec, each in a separate partition

Once the index file is opened, the header, ItemIDs list, and lookup structures are deserialized and kept in memory. In addition to the general index properties, the memory-mapped index file storage engine's header also stores the offsets to various data partitions in the file, such as the start of the serialized list of ItemIDs, or the used data layers. To optimize prefetching during a query, the index file is memory-mapped. This enables the operating system to transparently load subsequent file pages into memory while the query is being executed, thereby optimizing storage media access, I/O operations, and cache hits.

The Voxel Lookup Structure

The voxel codec part of the memory-mapped index file uses a 3D array-based page table, as described in the Spatial Indexing Section 6.2. This lookup consists of a 3D array where each coordinate holds the number of index items with a value at this coordinate, the memory offset to the data page in the index file, and the data page size. Without compression, the data page size can be calculated due to the fixed size of data entries and mapped ItemIDs in a voxel codec. However, once the inline compression, outlined in Section 7.4, is enabled, the data page size must be stored in the lookup, as it can no longer be calculated. Given that all lookup entries are of a fixed size, and not dependent on the amount of index items, the lookup size only depends on the reference space size. Each data layer requires a separate page table because the number of index items with data entries at each coordinate may vary across layers, as well as the compressed data page size of coordinates.

At high index resolutions, the 3D array lookup structure can consume a significant amount of memory. For example, a 1000 x 1000 x 1000 voxel resolution with an entry count size of 4 bytes and a file offset of 4 bytes requires approximately 8 GB of memory (without inline compression, thus without storing the size of the data pages). To mitigate the high memory usage, only pointers to entries are stored in the page table, allowing coordinate entries without actual data entry to be set to nil during runtime. However, it is still necessary to write the empty entries when serializing the lookup structure to the index file to ensure that the lookup can be deserialized correctly.

The Region Lookup

The region lookup of the memory-mapped file storage engine is composed of two parts, as also outlined in the Spatial Indexing Section 6.2. The first part is responsible for mapping the coordinates to regions, as described in Section 6.2.2. As with the lookup structure for a voxel codec index, the mapping of coordinates to regions is stored in a 3D array. Each coordinate holds the region ItemIDs that are assigned to that coordinate. The second component is a map that links region ItemIDs to corresponding region sample information, including the sample's ItemID, file offset, and data page size. These region and region sample lookup structures are serialized and stored within the index file.

Keeping the complete index item identifier (ItemID) along with the in-memory offset and size in the lookup can cause a memory bottleneck due to the large number of entries the lookup has to support. Unlike the voxel codec's lookup table, which is independent of the number of index items, the memory usage of the region lookup increases with the number of regions and region samples. Therefore, the usage of the mapped ItemIDs, explained in Section 7.2, is essential for reducing its memory requirements.

The memory mapping of the index file is efficient as long as the data required resides close to the currently read data in the file. Since the data is ordered according to the space-filling curve, memory mapping is very effective in allowing the operating system to transparently handle the loading of the required data into main memory ahead of its usage. The memory mapping's preference is the file data area after the currently accessed one. To prevent jumps between memory locations, SPX incorporates forward-read-flow synchronization, as detailed in Section 6.4.3.

7.7 Creating Large Indices - Implementation Details

The basics of the index creation are already outlined in Section 6.3. However, certain implementation details are not yet covered. As mentioned before, a new index is created by defining a reference space and selecting an appropriate index codec. The index reference space is traversed using a space-filling curve, ensuring locality and establishing data order. Depending on the codec type, region or voxel codec, the index data handling is different. The creation process initializes a data factory for each index item. All current data factory implementations keep the entire amount of data in memory for each index item, which limits the number of items that can be indexed.

To overcome this limitation, the final index must be built incrementally. The index items are divided into buckets, and indices are created containing only the items in each bucket. These indices are then successively merged into larger indices until the final index is created containing all index items. The number of indices to be merged in one run is limited by the memory footprint of the storage engine. Therefore, several incremental merges may be necessary to obtain the final index.

Voxel Indices - Implementation Details

Creating a voxel codec interface requires that the index item identification happens for each data entry at each coordinate. Using the complete ItemID would significantly increase the index size. Therefore, as explained in Section 7.2, SPX uses only the mapped ItemID for all internal purposes. Initially, during index creation, all ItemIDs are added to the ItemID list and replaced with their corresponding mapped ItemID (the list index). Instead of the large ItemID, the smaller mapped ItemID is used for all data entries in the voxel codec index.

The index's creation process involves the traversal of the reference space along the chosen space-filling curve, and the creation of a data page for each coordinate, consisting of the tuples of mapped ItemID and data entry of all index items having a data entry at the coordinate. The data page can optionally be compressed, as outlined in Section 7.4, prior to storage. Once a query result is obtained for a query, typically containing the mapped ItemIDs in combination with the respective result values, the mapped ItemIDs are replaced by the original ItemIDs, and the result is returned to the user. The index creation of a region codec based index contains similar implementation details.

Region Indices - Implementation Details

In the region codec's data creation, the first step is to create the data factory for each data collection. During the index creation, each region is handled only on its first occurrence during the reference space traversal along the space-filling curve. At this point, the region ItemID is added to the list of ItemIDs, explained in Section 7.2, and the region's ItemID is replaced by the mapped ItemID for all further references to the region. Also at this point, the data factory of the corresponding index item produces all region sample ItemIDs next to their data entries. Again, the sample ItemIDs are added to the list of ItemIDs, and the region sample's ItemID is replaced by the mapped ItemID, which is then used throughout all mappings, processing, and data handling, to identify the sample. Whenever a region ItemID or a sample ItemID occurs in a query result, the mapped ItemID is replaced by the original ItemID before the result is returned to the user. This replacement must be also done during the merging of indices, described in the next section.

7.8 Merging Indices

For indices to be merged, they must share the same reference space, have the same number of voxel and region data layers, and use the same codec. Other properties, such as the space-filling curve, the compression, or the storage engine can differ, and can also be freely chosen for the final index. The merging can merge several indices into one.

Each index is opened by its storage engine. During the merging process, the initial step is to merge the ItemID lists. All ItemIDs of each index involved must be remapped to newly mapped ItemIDs, outlined in Section 7.2. This is done by merging the ItemID lists, and the newly mapped ItemIDs are again the list indices of an ItemID. This remapping process also screens for duplicate entries among the indices. If a duplicate is found, the merging process is terminated to avoid data duplications.

As done during the creation process, the reference space is traversed along a chosen space-filling curve. The space-filling curve index is reverse-mapped to obtain the 3D coordinate, which is skipped if it falls outside of the coordinate space of the reference space of all indices. For indices with different space-filling curves, the reference space is traversed along the space-filling curve of the final index. Since the storage engines may use the space-filling curve index to access the data pages, each index with a different space-filling curve must maintain its own space-filling curve to map 3D reference space coordinates to their own 1D space-filling curve indices. The differences between the merging of voxel codec and region codec indices are explained in the following sections. In indices of codecs implementing both, the voxel and region interface, the voxel part is merged before the region part. In the next section, the respective merge details are provided, with the indices to be merged being referred to as partial indices.

Merging Voxel Indices

As the reference space is traversed along the space-filling curve of the final index, the data page for each coordinate is retrieved from each partial index and decomposed into tuples of mapped ItemIDs and data entries. Each mapped ItemID is used to retrieve the complete ItemID from the respective index's ItemID list, which is then remapped to its new mapped representation from the merged ItemID list. The new tuples, consisting of the remapped ItemID and the corresponding data entry, are then used to construct the data page for that coordinate in the final index. This process is repeated for every coordinate in the reference space and across all voxel data layers.

Algorithm 7.1: The merging of several vocel codec indices in basic form in pseudo-code.

Input: *ses*: a list of storage engines with opened indices
Input: *sfc*: the space-filling curve of the final index
Input: *dataLayers*: number of voxel data layers
Output: *mergedSE*: the storage engine of the merged final index

```
   // merge all ItemID lists and control for duplicates
 1 mergedIDList ← [];
 2 for i ← 0 to length(ses)-1 do
 3 |   forall ItemID in ses[i].GetItemIDs() do
   |   |   // add the ItemID to the merged list, check for duplicates
 4 |   |   if mergedIDList.contains(ItemID) error;
 5 |   |   mergedIDList.append(ItemID);
 6 |   end
 7 end
 8 mergedSE.setItemIDList(mergedIDList);
   // traverse along space-filling curve
 9 for idx ← 0 to sfc.getMaxIndex() do
10 |   x, y, z ← sfc.idxToPos3D(idx);
11 |   if outsideOfReferenceSpace(x, y, z) continue;
12 |   for l ← 0 to dataLayers-1 do
   |   |   // assemble the data page for this coordinate from all indices
13 |   |   mergedDataPage ← [];
14 |   |   for i ← 0 to length(ses)-1 do
15 |   |   |   dataPage ← ses[i].getVoxelData(l, x, y, z);
   |   |   |   // handle the compression of the sample data, save the data
16 |   |   |   if isCompressed(dataPage) dataPage ← decompress(dataPage);
17 |   |   |   while tuple extract from dataPage do
   |   |   |   |   // get the new mapped ItemID
18 |   |   |   |   oldItemID ← tuple.ItemID;
19 |   |   |   |   unmappedItemID ← ses[i].getUnmappedItemID(oldItemID);
20 |   |   |   |   newItemID ← mergedIDList.getIndex(unmappedItemID);
   |   |   |   |   // add the new tuple to the merged data page
21 |   |   |   |   mergedDataPage.append((newItemID, tuple.data));
22 |   |   |   end
23 |   |   end
24 |   |   mergedSE.setVoxelData(l, x, y, z, idx, mergedDataPage);
25 |   end
26 end
   // the storage engine is the container of the index
27 return mergedSE;
```

Merging Region Indices

The merging of region indices follows the order of the final index's space-filling curve. During the traversal of this curve, the unmapped 3D coordinates are used to retrieve the regions from each partial index. The combined list of regions of all partial indices is the region list of the current coordinate in the final index's coordinate-to-regions lookup. If the region appears for the first time during the traversal, its complete block of region sample data including the sample ItemIDs for that region is requested from the respective partial index. Since the merged indices use the same codec, there is no need to decode the sample data and it can simply be copied over. However, if compression is enabled and different compression methods are used, decompression and recompression are required. The region sample data is stored in the final index's region data layers. Nevertheless, to obtain the right access information, each sample still requires to be written separately to the final index. Also, the sample ItemIDs are required to be remapped to their new mapped ItemIDs, which are in turn added to the region-to-samples lookup in combination with the respective sample's access information as a tuple in the final index. This only happens for regions that have not yet been stored in the final index. Algorithm 7.2 outlines the merging of region codec indices in a simplified form.

Algorithm 7.2: The merging of several region codec indices in basic form in pseudo-code.

Input: *ses*: a list of storage engines with opened indices
Input: *sfc*: the space-filling curve of the final index
Input: *dataLayers*: number of region data layers
Output: *mergedSE*: the storage engine of the merged final index

```
// merge all ItemID lists and throw an error if duplicates are found (see the
   voxel codec merging algorithm)
1  mergedIDList ← mergeItemIDLists(ses);
2  mergedSE.setItemIDList(mergedIDList);
3  regionsSaved ← [];
4  for idx ← 0 to sfc.getMaxIndex() do
5      x, y, z ← sfc.idxToPos3D(idx);
6      if outsideOfReferenceSpace(x, y, z) continue;
7      regionIDsOfCoordinate ← [];
8      for i ← 0 to length(ses)-1 do
9          regionIDs ← ses[i].getRegionIDsAtCoordinate(x, y, z);
10         forall regionID in regionIDs do
               // get or create the new mapped region ID
11             unmappedRegionID ← ses[i].getUnmappedRegionID(regionID);
12             newRegionID ← mergedIDList.getIndex(unmappedRegionID);
13             regionIDsOfCoordinate.append(newRegionID);
               // save each region only once
14             if regionsSaved.contains(regionID) continue;
               // save all the region samples of the region
15             sampleIDs ← ses[i].getRegionSampleIDs(regionID);
16             forall sampleID in sampleIDs do
                   // create the new mapped sample ID
17                 unmappedSampleID ← ses[i].getUnmappedSampleID(sampleID);
18                 newSampleID ← mergedIDList.getIndex(unmappedSampleID);
19                 for l ← 0 to dataLayers-1 do
20                     sData ← ses[i].getRegionSampleData(sampleID);
                       // handle the compression of the sample data, save the data
21                     if isCompressed(sData) sData ← decompress(sData);
22                     mergedSE.setSampleData(l, newRegionID, newSampleID, sData);
23                 end
24             end
25             regionsSaved.append(regionID);
26         end
27     end
28     mergedSE.setRegionsAtCoordinate(x, y, z, idx, regionIDsOfCoordinate);
29 end
   // the storage engine is the container of the index
30 return mergedSE;
```

CHAPTER 8

Use Cases & Applications

In this chapter, I discuss the two executables of the SPX framework, SPI, the index generation and maintenance tool, and the SPX daemon, which provides the index querying service. This is followed by a description of the applications BrainBaseWeb and BrainTrawler, which are currently utilizing the SPX framework. Each application introduction is followed by a description of the codecs used, including the queries they support.

8.1 The SPX Executables

The SPX framework exposes its functionalities via several packages. The SPX executables build upon these packages and provide the necessary interfaces for managing the indices and querying them.

SPI - Index Creation and Maintenance

SPI serves as the primary tool for managing various tasks related to the indices, including their creation and their merging. SPI was the result of extracting the command-line interface from the prototype developed by Florian Schulze and extending it to support the new codec interfaces, query areas, storage engines, and ItemIDs. SPI parses the command-line arguments, checks their validity, and initializes the given tasks. It can create indices, merge indices, inspect the stored index data, and execute queries.

For index creation, SPI initializes the codec's data factories and oversees the index creation process, outlined in Section 6.3. During the merging of indices, it opens all indices and validates if they can be merged, as well as the space-filling curve and the storage engine of the resulting merged index. Details of the merging process can be found in Section 7.8.

Additionally, SPI is equipped with diagnostic tools. It can inspect the index items by requesting the list of ItemIDs of an index, and obtain basic index properties by accessing the header of the index. It can inspect payloads for a given coordinate, monitor memory consumption, and evaluate the processing time of different parts of SPX using profiling tools. Furthermore, it can execute queries by specifying the index, the type of query, and the associated initialization parameters. The results can be displayed directly on the command line or exported to a JSON file. Nevertheless, SPI is an executable, that is invoked, does its task, and exits. It does not provide a service interface for continuous querying. This is the task of the SPX daemon.

SPX Daemon - Query Service Provider

The SPX daemon operates as a service provider for existing indices and represents the primary access point for querying. Upon startup, it opens and initializes all indices specified by command-line parameters. It offers a REST interface, supplying information about the indices it manages, and establishes a separate subroute for each index to obtain an index's general properties derived from its header, outlined in Section 7.1. Another endpoint of each opened index serves as a receiver for query requests. Responses to these queries are encoded in JSON format and returned to the requesting instance. An initial version of the SPX Daemon, which included basic index handling and a query service, was part of Florian Schulze's prototype. The author expanded the SPX Daemon by adding support for various index types and codecs, as well as extending the range of query types, including multi-query capabilities. The SPX Daemon handles all client communication in the JSON format to avoid any dependencies on the client's programming language.

8.2 The Brain* Framework

The Brain* framework [85] is a collection of tools and applications developed by the VRVis research center to support neurobiologists in their research. It is designed to facilitate the exploration of large-scale brain data, such as gene expression data, confocal microscopy images, and segmented structures. The framework is used in several projects, including the LarvalBrain project [86] and BrainTrawler [25, 27]. Its goal is to provide a foundation to rapidly develop new applications for neurobiologists.

8.3 BrainBaseWeb - Larvalbrain

BrainBaseWeb is a web-based application developed as part of the LarvalBrain project [86] at the VRVis research center in cooperation with the workgroup of Prof. Dr. Andreas Thum of the Department of Genetics at the University of Leipzig. It serves as a resource for exploring GAL4/UAS single-channel confocal microscopy images and collections of segmented structures including axon tracts and neuropils of the Drosophila melanogaster. At the time of writing, it contains thousands of single-channel images of Drosophila

melanogaster in the L3 larval stage, with their driver and reporter information, registered onto a template built by Münzing et al. [60].

SPX in BrainBaseWeb

Within BrainBaseWeb and the LarvalBrain project, SPX is used to index the single-channel GAL4/UAS staining images, and the available segmented structures. Indices based on the *staining codec* [76], the *distance-field codec* [12, 79], and the *structure codec* [26] are used to spatially query the image data available with queries made possible by the codecs. A separate section in BrainBaseWeb provides the possibility for spatial exploration of the included data.

Spatial Queries

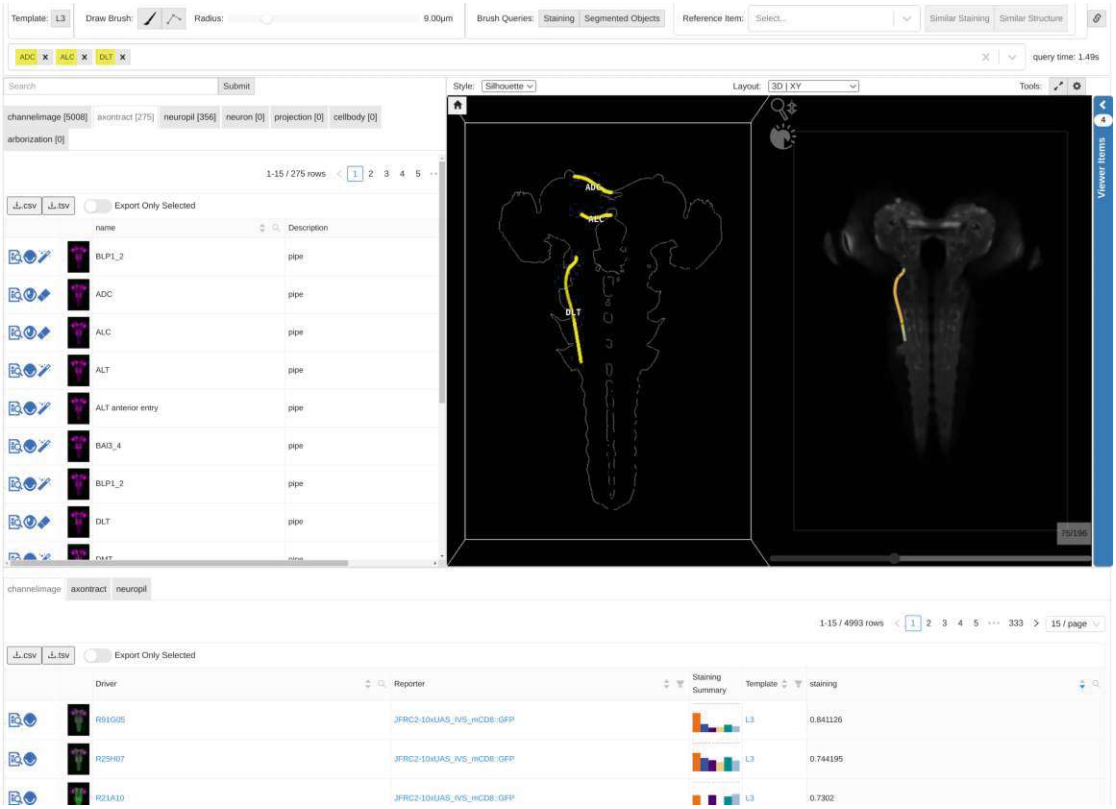

Figure 8.1: BrainBaseWeb's spatial query interface. The user can select a region of interest by brushing in the slice rendering. The query results are displayed in a table. In this example, a brush was created with the help of the axon tracts skeleton graph data.

In the Spatial Query Section of BrainBaseWeb, the user can create a query area by either brushing in a slice rendering, which shows the registration template overlaid with selected

single-channel images and segmented structures, or by using the magic brush function. The magic brush function automatically creates brushes from segmentations saved in skeleton graphs and masks for label files of segmented structures. An explanation of brushes and masks can be found in Section 7.3. Any combination of brushes and masks can be used to create a query area, which is used in one of the available query types. The queries are mapped by their type to the spatial indices, based on the codec able to handle this query type. The execution is triggered using the SPX daemon's REST interface. The query request is sent from BrainBaseWeb to an API service, which forwards it to the SPX daemon. Once the query results—tuples of ItemIDs and result values—are returned from the SPX daemon, the API service supplements the results with additional information from the relational database. Finally, query results are displayed in a table in BrainBaseWeb. An example can be seen in Figure 8.1.

Currently, BrainBaseWeb offers three types of indices and queries based on the codecs used for the indices. The codecs and queries are explained in the coming sections. The following indices are available in BrainBaseWeb:

- **Staining Index** - based on the *staining codec*, offering the *high-staining query* and the *similar-staining query*.
- **Object Index** - an index using the *distance-field codec*, providing the *object query* to find segmented instances of neurological structures.
- **Structure Index** - using the *structure codec*'s *similar-structure query* to find images or axon tracts with similar structural properties to a reference entity.

Often researchers are interested in combinations of spatial queries. At the time of writing, the public version of BrainBaseWeb only supports one spatial query result at a time. The anatomical queries were introduced to alleviate this limitation, before the multi-query system, outlined in Section 6.4.5, was introduced into SPX. The anatomical queries are explained in the next section.

Anatomical Queries

The second index application within BrainBaseWeb involves anatomical queries, in which cached profiles of relative staining values of single-channel images for each neuropil are used to find images with certain staining properties. The staining profiles, created for each neuropil segmentation using high-staining queries executed with a mask query area, contain the relative staining values of each image in the database and are stored in the relational database of BrainBaseWeb. The user selects neuropil segmentations, and the cached profiles are loaded from the database for the selected segmentations. A profile contains the relative staining values of each image in the database for the selected neuropil segmentation. Each axis in a parallel coordinate system represents one of the selected neuropils. This enables the user to find images with specific staining properties across a combination of the selected neuropils. Similar profiles are created for axon tracts using

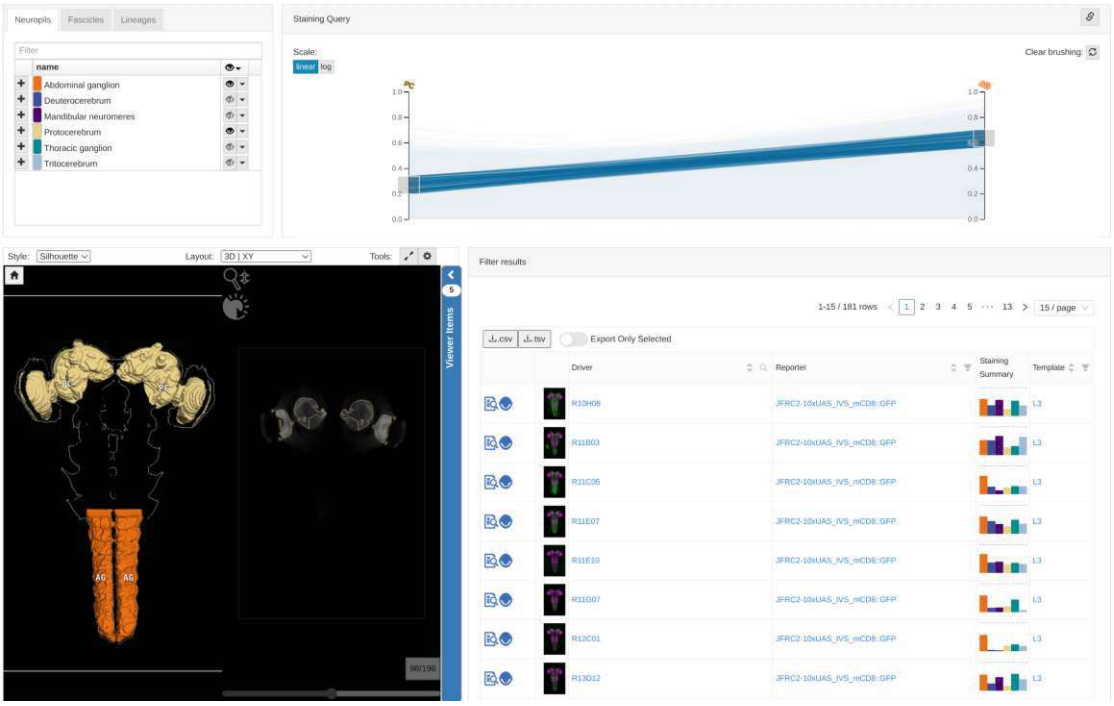

Figure 8.2: BrainBaseWeb's anatomical queries - expression strength filtering in neuropil areas with parallel coordinates.

the structure index and the similar-structure query. The concept of anatomical queries was established as soon as the staining and structure codecs were ready. At the time, BrainBaseWeb could only handle one spatial index query result, and anatomical queries addressed this limitation by caching the results in the database. Figure 8.2 shows the anatomical query interface of BrainBaseWeb.

8.4 Codecs in BrainBaseWeb

In BrainBaseWeb, indices based on the *staining codec*, the *distance-field codec*, and the *structure codec* are used to spatially query the available image data. Due to the large volume of data, building these indices required several intermediate steps. Each implemented data factory keeps the entire image data in memory, which limits the amount of data that can be indexed in a single run. The indices are built in several steps, with each step indexing a subset of the data. The final index is created by merging the subset indices, as explained in Section 7.8. The used codecs are outlined in the context of Larvalbrain and BrainBaseWeb in the following sections.

8.4.1 The Staining Codec

The staining codec, introduced by Schulze [76], is a voxel codec that facilitates the identification of confocal microscopy single-channel images with high neuronal staining in a given query region. The implementation of this codec was extracted from Schulze's prototype at the project start and restructured to adhere to the voxel codec interface outlined in Section 5.3.1. Basic multi-routine support was added through the use of a Go channel. When a worker routine of SPX's multi-routine query engine calls the data-handling function of the query processor to process the data entries of a coordinate's data page, the data is inserted into the channel. A separate Go routine then takes the data from the channel and aggregates it subsequently. Test results showed that using an unbuffered channel and a separate Go routine for data aggregation delivered better performance than using synchronization primitives like mutexes to make the result aggregation thread-safe.

Data Sources from BrainBaseWeb

Single-channel confocal microscopy images showing the reporter channel, and thus the targeted genes expressed, are the prevalent data source for the staining codec. These images often exhibit high noise levels, necessitating a denoising process. This process is achieved through thresholding using a histogram-based binning method, retaining approximately the top 1% of voxels, followed by a binary opening to strengthen structural expressions and further reduce noise. The remaining voxels form a binary mask which serves as the index data.

Segmented structures are integrated into the binary mask concept by using binary label files containing one or multiple structure segmentations as binary masks. These label files are either available from the segmentations done by biologists, or created using splatting algorithms in case of pipe-like structures stored as skeleton graphs, or polygonal meshes. For skeleton graphs and meshes, additional preprocessing is necessary to derive binary masks of the structures. The current implementation of the staining codec does not do the preprocessing of images and segmentations into the staining masks. An external tool was used to do the preprocessing, and the codec's data factories only loaded the binary mask files.

Data Generation

Each binary staining mask file is handed over to the indexing tool SPI, outlined in Section 8.1, along with its serialized item identifier explained in Section 5.1. The indexing tool then instantiates a staining codec factory for each mask, loading them completely into memory. The staining codec's data factory encodes the mask into an 8-neighborhood bitfield. Always eight voxels are represented by one byte, and the bits in the byte are masked if a coordinate is marked in the staining mask. Each coordinate is represented by a bit, which effectively halves the reference space in all dimensions.

Codec Queries and Use Cases

Schulze et al. [76] use the staining codec to provide the following query options:

High-Staining Queries

The high-staining query yields ItemIDs of images and segmentations, where staining was detected within the coordinates of the query area. This means it identifies images having the target genes strongly expressed, and segmentations of structures, by finding images with a high number of voxels flagged in the binary mask data. The returned relative staining value signifies the proportion of the query region with high staining in a given image. This query serves as an effective approach for locating images where area-based neuronal structures are distinctly visible. Given the image registration, such structures tend to appear approximately at the same locations across all images. However, this query can only detect structures and staining within the query region.

The Similar-Staining Query

This query requires a reference image or segmented structure in addition to the query region. The current implementation necessitates that the reference item is included among the index items, thereby ensuring the presence of comparable data within the query areas. The query only conducts comparisons for coordinates where the reference index item exhibits staining. The result is a list of images or segmented structures, each associated with a similarity value based on staining as compared to the staining of the reference item. If several images have similar staining expressed in a region of interest, it strongly indicates that the images show the same structural entity within the query region. Schulze et al. [76] define the similarity measurement s as the Dice coefficient, which considers the binary staining masks of index items i and j within the query area R. In this context, σ symbolizes the indexing function:

$$s_{t,a}^{R} = \frac{2 \sum_{\vec{v} \in R} \sigma(i, \vec{v}) \cdot \sigma(j, \vec{v})}{\sum_{\vec{v} \in R} \sigma(i, \vec{v}) + \sum_{\vec{v} \in R} \sigma(j, \vec{v})} \tag{8.1}$$

8.4.2 The Distance-Field Codec

The distance-field codec, also a voxel codec, enables the identification of segmented structures within a query region by querying a distance-field index. It helps identify segmentations that intersect with or are in close proximity to the area of interest. Solteszova et al. [79] introduced this codec to locate structures in Drosophila Larvae in the L1 stage within query regions, allowing for the interactive highlighting of the structure's mesh in the rendering. Schulze et al. [76] utilized it as well to query objects around a point in space using the term object-distance index. The implementation of the result aggregation follows the same operating principle as the staining codec, using a Go channel and a separate Go routine for data aggregation.

Data Sources

Biologists annotate, identify, and label structures in volumetric image stacks. The type of data saved depends on the type and shape of the segmented data. Tubular structures, such as segmentations of axon tracts or projections, are stored as skeleton graph files containing information about key points and connections between them. Regional data are saved as polygonal meshes, either created during segmentation or based on binary label files generated during segmentation. The index data generation is based on binary mask volume files, with one binary mask file created for each segmented object. These binary mask files are the same as those used in the extension of the staining codec to add segmentations into a staining index. In contrast to the previously described staining codec, indices based on the distance-field codec only contain segmented structures.

Data Generation

During the preprocessing stage, the distance-field codec initializes a payload factory for each binary mask file and generates a signed Euclidean distance-field. This involves a three-step process. Firstly, a Euclidean distance-field is created using the method described by Maurer [50]. Secondly, a distance-field is created from the inverted mask image, capturing the distances inside the segmented structure. In the final step, the inverted distance-field is subtracted from the original distance-field, generating negative distances inside the structures and positive distances outside for each voxel. The codec's data factory also applies a distance cutoff, setting distances far from the structure to an infinity identifier (a special marker indicating that the distance exceeds a certain threshold) and excluding them from the indexed payload sets of a voxel located outside the cutoff distance.

Queries and Use Cases

Schulze et al. [76] defined the following general use cases for the staining index:

Anatomically motivated spatial exploration of available objects:
The index enables the user to locate objects close to the query area or intersecting it.

Finding already annotated objects related to unknown structures in a given channel image:
The user highlights an observed or suspected structure in a channel image and uses the distance-field index to find already known and segmented structures.

Exploring the spatial relatedness of objects:
The index can be used to find objects close to each other or overlapping, indicating structural connectivity.

Deriving functional information:
The overlaps of objects (e.g. arborizations and neuropils) in different brain regions indicate the functions of the neurons.

The Object Query

The object query returns a list of ItemIDs that either overlap with or are in the local vicinity of the brush query. The result value for each item can be positive or negative, and its interpretation depends on the sign. Positive values indicate the minimum distance from a segmented structure to the query region, implying that the structure is outside the query region. Higher values mean that the closest point of the structure to the query area is farther away, while lower values indicate that the structure is closer. Only index items that fall within the distance cutoff set during index creation are returned. If the value is negative, it indicates that the structure is overlapping with or inside the query region. In this case, the query returns the count of voxels inside the query region, to enable an overlap estimation.

The Object-Overlap Query

This reference object query returns the proportion of overlap between the reference object and other objects within the query area. Further information, including a formal definition of the overlap, can be found in Schulze et al.'s report [76].

8.4.3 The Structure Codec

Biological structures are not always accurately placed in the same locations inside different brains. Finer structures can be difficult to co-locate in registered images of different samples. The structure codec facilitates this by creating gradient vector flow fields of images [26]. The gradient vector flow field is a 3D vector field that represents the strongest gradient vector at each voxel in the image. The similarity of these gradient vector flow fields is compared to find images containing the same structures.

Data Sources

The structure codec primarily indexes single-channel images and finely segmented tubular structures, such as segmented instances of axon tracts and projections stored as skeleton graphs. In its current state, it is not well-suited for area-based structures such as neuropils and arborizations. This limitation stems from the process of vector flow field creation, which generates an undesired 'black hole' effect—an area with no gradient vectors—within area-based structures.

Data Generation

Several key factors play a significant role in influencing the performance of a structure index. Higher resolution allows for finer detection of structures in the indexed images. Given the significant noise level in the indexed single-channel images, which could disrupt the vector flow field, it is necessary to incorporate a preprocessing step that employs a Gaussian convolution filter for denoising purposes. However, it is crucial to achieve a balance in the degree of smoothing applied by the Gaussian convolution filter to avoid losing finer structures. In the context of splatted skeleton graph structures, the diameter of the splatted pipes needs to exceed one voxel to yield a gradient vector flow field of practical use.

The tasks of preprocessing and gradient vector flow generation are managed by an external preprocessing tool developed by Ganglberger et al. [26]. The results are files in which each coordinate corresponds to a 3D vector containing the strongest gradient vector encoded as one byte for each dimension. The structure codec initializes a data factory for each index item, which loads the gradient vector flow field file and returns the vector for a voxel coordinate upon request during the index creation.

Queries and Use Cases

The use case for the structure codec is to find images containing the same structures as those in a provided reference image or segmented tubular structure in the index. Since the similarities of the gradient vector flow fields are compared, the structure can be spatially shifted and outside the brush area and still be detected. For the similar-structure query to be successful, the query area must be larger than the searched structure to contain a significant amount of the gradient vector flow field around it.

The Similar-Structure Query

The similar-structure query takes a reference image or segmented structural tubular object and compares the 3D vector flow field of the gradient vector flow. It returns the sum of squared differences between the gradient vector flow fields of the reference entry and the indexed entry of all voxels in the query area. The resulting similarity values are average gradient vector similarities, which are thresholded to remove low similarity values resulting from coincidental random noise gradient vector similarities. The returned result is a list of image ItemIDs and segmentation ItemIDs with their relative similarity value to the reference index item.

8.5 BrainTrawler

The second application using the SPX framework, BrainTrawler, was developed by Ganglberger et al. [25, 27] in cooperation with the Haubensack group at the Vienna Institute of Pathology (IMP) and Boehringer-Ingelheim, Global Computational Biology and Data Sciences Department. BrainTrawler is a visual analytics resource and tool for the iterative exploration of heterogeneous, large-scale brain data, built on top of the Brain* framework 8.2. It is designed to support biologists in the exploration of brain connectivity data and gene expression data.

The SPX Framework in BrainTrawler

BrainTrawler uses the SPX framework in the gene expression exploration part, while the connectivity data part [24] could potentially be integrated into SPX as a new codec. The data integration of BrainTrawler inserts data into two kinds of spatial indices: one utilizing the *gene-expression-value codec*—a voxel codec-and one based on the *gene-sample-meta codec*—a region codec. Both codecs are explained in the next sections. The combination of several datasets mapped onto a common reference space, including voxel grid-based gene expression data, connectivity data, and single-cell RNA transcriptomics sample data, opens up new avenues for knowledge discovery on a larger scale.

Querying Volume Based Gene Expression Data

The *gene-expression-value codec* indexes dataset-normalized voxel grid-based gene expression data mapped onto a common reference space (e.g., the Allen Mouse Brain [40]) for all integrated datasets. Similar to BrainBaseWeb, BrainTrawler enables users to brush arbitrary query regions, use predefined brain region masks, or combine both methods for creating query regions. The queries are sent to the SPX daemon, and the results consist of the average gene expressions in the query region. The results are then supplemented by the BrainTrawler API with additional information, and displayed in BrainTrawler in a table. Figure 8.3 shows an example. The results are automatically sorted by the average gene expression value.

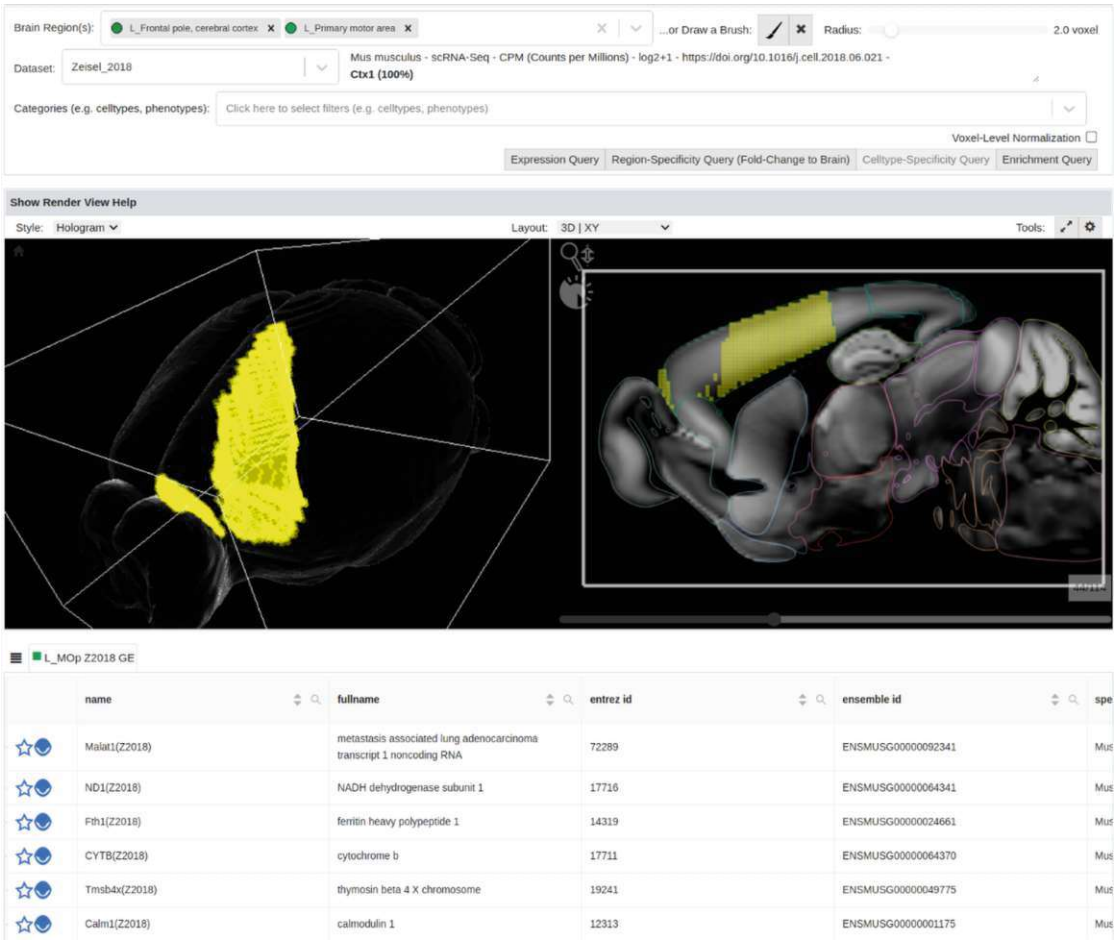

Figure 8.3: Querying volume-based gene expression data in BrainTrawler. - *Image from [25, 27]*

Exploration of Brain Region Level Gene Expression

BrainTrawler allows users to explore gene expression data at the brain region level, organized by datasets and different aggregation categories such as cell type, genotype, or phenotype. The *gene-sample-meta codec* is used to index single-cell RNA samples associated with one or multiple brain regions in the reference space. Each sample has a set of metadata properties, including cell type, genotype, phenotype, age category, or strain, along with the observed gene expressions. SPX uses this index to make the sample data searchable and filterable by metadata properties, and it aggregates the gene expressions using different user-chosen aggregation categories on the fly. The aggregated categorized gene expressions are displayed in a heatmap, as seen in Figure 8.4, where the categorization is based on the cell type of samples. The rows represent the different cell types in the selected datasets, while the columns represent the brain regions. The heatmap displays the average gene expression values for each cell type in each brain region. The user can hover over the heatmap to see the gene expression value, the

categorization value, and the brain region.

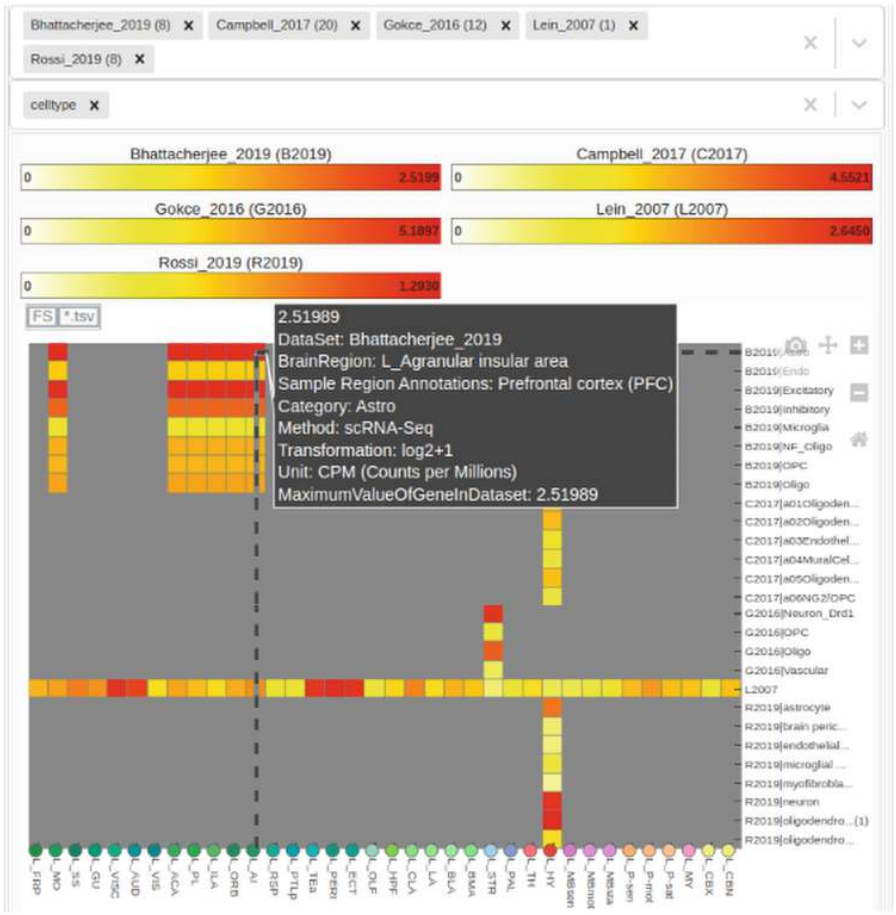

Figure 8.4: An example for a region-level gene expression heatmap of different datasets with cell type category aggregation. - *Image from [25, 27]*

8.6 BrainTrawler Lite

BrainTrawler Lite is an extension of BrainTrawler designed to ease the workflow of one of its main research goals: quickly finding and observing expressions of certain genes in brain regions across multiple published datasets. The datasets consist of single-cell sample data, including gene expression matrices that can be tens or hundreds of gigabytes in size and originate from thousands of cell samples.

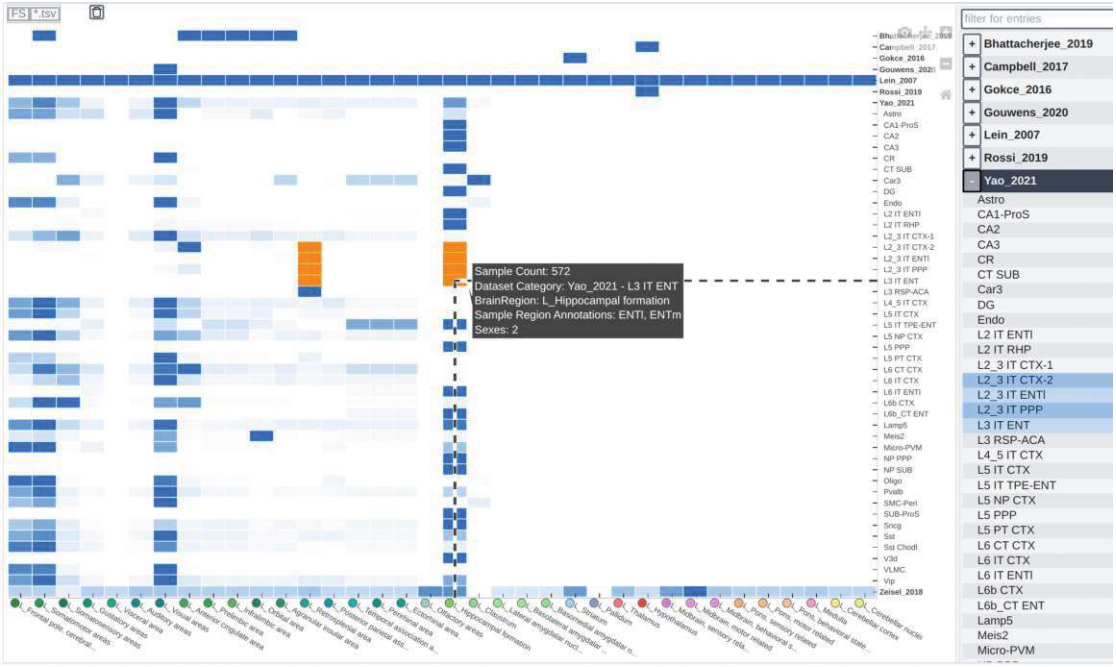

Figure 8.5: The BrainTrawler Lite overview heatmap for the mouse brain shows the distribution of cell samples across different region samples available for datasets or categorical subgroups. The saturation of blue indicates the amount of cell samples in a region, for a dataset or categorical subgroup. Orange highlights the selected brain region and dataset combinations used in the following workflows. - *Image from [25, 27]*

Cell samples, that belong to specific cell types (e.g., Astrocyte, Neuron), may have associated metadata such as genotype, phenotype, age category, and strain. These samples are stored as region samples in a *gene-sample-meta codec*-based index, with the regions corresponding to brain regions in the common reference frame. The *get aggregated* query, described in the next section, aggregates the cell samples by categorical class within a dataset and brain region to obtain a mean expression value. The aggregation categories, chosen by the user, group the samples according to their dataset and categorical values. This aggregation occurs in real-time as the sample data entries are processed in the index, with all data in BrainTrawler Lite being directly derived from the index. An overview heatmap, shown in Figure 8.5, displays the number of samples available in each dataset

and categorical group for each brain region.

BrainTrawler Lite offers two workflows: *expression of genes* and *genome-level analysis*. In both cases, the basis of the observation is a collection of tuples of brain regions and datasets' categorization value subgroups, chosen by the user in the overview heatmap seen in Figure 8.5.

Expressions of Genes

In this workflow, the user is interested in specific genes or collections of genes called gene ontology terms [6, 16]. Querying the gene-sample-meta index includes filtering the gene expression data to only aggregate the selected genes. The goal is to display a comparative visualization using small multiples of heatmaps showing gene expression strengths, as seen in Figure 8.6, to identify patterns in brain region/cell-type combinations with strongly expressed genes.

Figure 8.6: Expressions of Genes - gene selection on top, dataset normalized color scales in the middle, small multiples heatmaps on the bottom right, with the detail view of the selected small multiple on the left. - *Image from [25, 27]*

Queries handling dataset-level aggregation and queries handling categorical sub-dataset categorization value-wise aggregations are evaluated directly in SPX by the *get-aggregated query* using the multi-query approach outlined in Section 6.4.5. Each change in the chosen dataset, categorization values, or selection of genes or gene ontology terms triggers a new query to the gene-sample-meta index, which is processed by an SPX daemon.

BrainTrawler Lite offers three views of the aggregated values, in which the first heatmap always shows the average expression strength. The user can display the expression value,

categorization value, and brain region by hovering over the heatmap area with the cursor.

- **Gene View** - This view shows one heatmap for each gene. The rows represent the datasets and categorization values, while the columns represent the brain regions.
- **Brain Region View** - This view displays each brain region separately as a heatmap of genes over datasets and categories.
- **Categorization View** - This view shows each chosen dataset or categorical value as a separate heatmap with rows representing genes and columns representing brain regions.

This enables biologists to observe the strengths and weaknesses of gene expression in different brain regions, categorized according to metadata properties across various publicly available resources. While in this workflow, the user needs to select the genes or gene ontology terms, in the genome-level analysis workflow, the user finds genes using the gene-sample-meta index.

Genome-Level Analysis

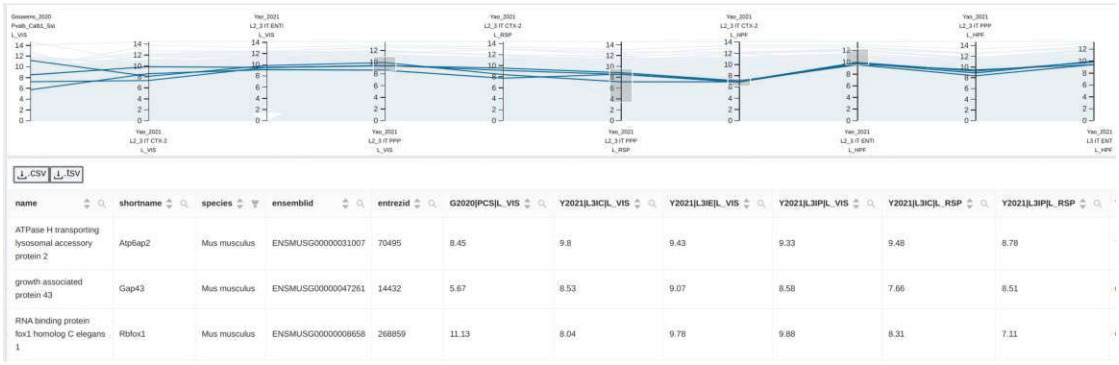

Figure 8.7: Genome level analysis - parallel coordinates of gene expressions, each axis is a combination of brain region and dataset/categorization value. The resulting list displays genes that match the brushed expression strength. - *Image from [25, 27]*

In the genome-level analysis, the user uses selections of datasets, categorization values, and brain regions in the overview heatmap to identify genes of a certain expression level in the chosen brain regions and categorization values. This is done by brushing gene expression strengths on the axes of parallel coordinates, where each axis represents a selection tuple—a combination of one brain region with one dataset or subcategory.

The result is a list of genes that fit the brushed expression value areas, showing their expression values according to the user's brushed regions on the parallel coordinate axis, as seen in Figure 8.7.

8.7 Codecs in BrainTrawler

8.7.1 The Gene-Expression-Value Codec

Ganglberger et al. utilized gene expression value indices to conduct queries for average gene expression distribution in query areas within BrainTrawler [25]. The indices facilitate the efficient and prompt aggregation of gigabytes of gene expression data in the reference space. Despite its application context suggesting otherwise, the gene-expression-value codec is actually the simplest codec. This is because any required preprocessing is performed outside the index's codec implementation, which solely reads and sets the coordinate value for each indexed item—in this case, the expression value of a gene.

Data Sources and Generation

The gene expressions originate from different published datasets; the measurements are stored in various file formats and coordinate spaces, which are not directly comparable. During BrainTrawler's integration pipeline, the different coordinate spaces are mapped onto a common reference space, such as the Allen mouse brain [40] or the Allen human brain [40]. The gene expression values are normalized to a one-byte value for each voxel and saved to a file, which is loaded into memory by a codec's data factory during the indexing process. The normalized gene expression values are directly stored in the index as data entries. Due to system memory limitations, building the final index is split into multiple steps, where indices with fewer datasets are built and merged later.

Queries and Use Cases

BrainTrawler uses the normalized and indexed gene expression values to quickly calculate the average gene expressions in a query region. The gene expression values are used to compare the gene expression values of different regions and across different datasets.

The Get Average Expression Query

This query returns the average gene expression values of a gene within the query region. The result is a list of genes with their average gene expression values for the indexed datasets.

8.7.2 The Gene-Sample-Meta Codec

The gene-sample-meta codec is a region codec that facilitates querying dataset-based collections of single-cell RNA samples representing a brain region. The codec enables querying the single-cell data based on user-defined query regions and directly via the brain regions. Each brain region has multiple single-cell samples associated with it. A sample's data consists of metadata properties, such as cell type, genotype, phenotype, age category, and strain, along with the observed gene expressions.

The gene-sample-meta index for BrainTrawler makes this data accessible in two data layers. To avoid memory congestion, the metadata and gene expression profiles are stored in separate layers. A gene expression profile is only read if necessary, based on the metadata of its sample. This two-layer approach enables SPX to avoid reading the gene expression data of samples that do not need to be included in the result sample set based on their meta properties, which is especially useful when dealing with large datasets, and large amounts of gene expression data. Each region and each sample is indexed individually, thereby offering high data access flexibility. In BrainTrawler Lite, and in parts of BrainTrawler, the used gene-sample-meta index is accessed using the region ItemIDs directly, instead of using a query area.

Data Sources

The gene-sample-meta codec sources its data from collections of single-cell samples obtained from biopsy sites in various brain regions. The single-cell data represents its originating region, and the exact biopsy site coordinates in the reference space are not typically available. Each sample in the data collection, or dataset, is associated with two types of tabular data. The first type encompasses the sample's metadata, such as sex, age, sample type, phenotype, etc. The second type of data is the gene expression profile for the sample, which lists the genes and their corresponding expression values. The metadata fields that define the actual data's shape, as well as the list of genes, are required to be consistent across all samples within a dataset, to ensure a correct encoding. The regions of a dataset are mapped into the reference space of the index, as described by Ganglberger et al. [25].

Data Generation

In the current implementation, the preprocessed data for each dataset resides in a folder named after the dataset. The sample metadata collection and gene expression profiles are stored as CSV files, including each sample's assigned region. Additionally, a mapping of reference space coordinates to regions is required to create the initial spatial indexing and provide spatial context for the data. The codec's data factory implementation loads the dataset's information by reading the provided CSV files for the sample's metadata, gene expression data, and the reference space coordinate to regions mapping. Each dataset is handled by a separate data factory instance.

The CSV files are parsed and stored within the factory instance in two data structures: one for the sample's metadata and one for the gene expression profiles. Both data structures contain a list of fields—metadata fields include entries such as sex, phenotype, genotype, etc., while gene expression profile fields include the list of genes. These two data structures form the basis of the two data layers. This enables SPX to store the metadata values of the samples in layer 1, while the gene expression values are kept in layer 2 and only accessed if necessary.

Queries and Use Cases

As with the previously described codecs, one way to query the gene-sample-meta codec is by defining a query region, which is mapped to overlapping regions. Queries using this type of access to the region information also provide the number of voxels in the processed region and the total number of voxels in the whole query region, which can be used for various purposes, such as a ratio of the query area overlapping a region compared to the full region extent. The second method of access is to query the regions directly, as well as individual samples. Depending on the query, extensive filter options are available to filter the regions based on their dataset and to filter samples by their metadata, thereby avoiding unnecessary reads of gene expression profiles. The gene expression data read for the remaining samples is directly processed and aggregated by the query handler function. This avoids the need to keep gigabytes of gene expression data in memory. The codec is used to explore and analyze gene expressions in different regions of the brain in relation to cell type and other categorizations across multiple published datasets. BrainTrawler Lite [27] operates purely on this codec.

The Get Aggregated Query

This query helps to obtain the averaged gene expressions for nested user-specified categories. The categories correspond to metadata attributes and are nested according to their values, the categorization values, which can also be specified to filter out samples not required in the result set. The result is a list of genes along with their category-wise averaged expression values. This type of query is useful for providing an overview of the gene expression profile within one or more regions, datasets, sample types, and sample properties. For instance, a categorical definition could be *cell type - sex* which would retrieve nested averaged gene expressions categorized by the cell type and sex of the samples. An example result might be: *Astrocyte - male - average gene expressions within this category.*

CHAPTER 9

Evaluation

This chapter evaluates the performance of the SPX framework, focusing on several key influencing factors. An architectural goal of the SPX framework was to enable it to scale effectively with hardware improvements. This chapter assesses various aspects of the SPX framework, including space-filling curve indexing, storage engines, inline compression, implemented codecs' query performance, and result encoding. The evaluation is divided into two parts: the first part examines the individual components of the SPX framework, while the second part assesses the performance of the entire query pipeline. In both parts, the results were recorded in CSV files and analyzed using diagrams created in Jupyter notebooks with Seaborn and Matplotlib [37, 45, 87]. The handling of the data was performed using Pandas [52, 80].

9.1 Hardware Setup

The performance evaluation was conducted on a notebook equipped with an Intel i9-13900HX CPU, 32 GB of RAM, and a high-performance 1 TB Kingston FURY Renegade PCIe 4.0 NVMe M.2 SSD featuring an onboard 369 GB SLC cache and 1 GB of DRAM cache. The operating system was NixOS with Linux Kernel 6.6, utilizing a Btrfs file system. All benchmarks and tests were conducted on Go version 1.21. It is important to note that full disk encryption using AES-XTS was enabled on the SSD, which may have marginally decreased the performance of the index file engine due to the additional overhead from encrypting and decrypting data.

9.2 Evaluation of Parts

The evaluation of the different parts of the SPX framework is done by measuring the performance of the different parts in isolation with the help of the Go benchmark tools. Measurements were made in nanoseconds and converted to milliseconds for better readability. All indices were constructed with a volume of 100x100x100 voxels, and the tests were conducted using synthetic data to ensure consistent results.

9.2.1 Space-Filling Curve Indexing

The framework enables the storage engine to use two lookup methods: direct query coordinate-based lookup and 1D mapping of coordinates via the space-filling curve. Consequently, the efficiency of converting 3D coordinates to a 1D space-filling curve representation becomes a performance factor to examine, particularly if handling a large number of query coordinates.

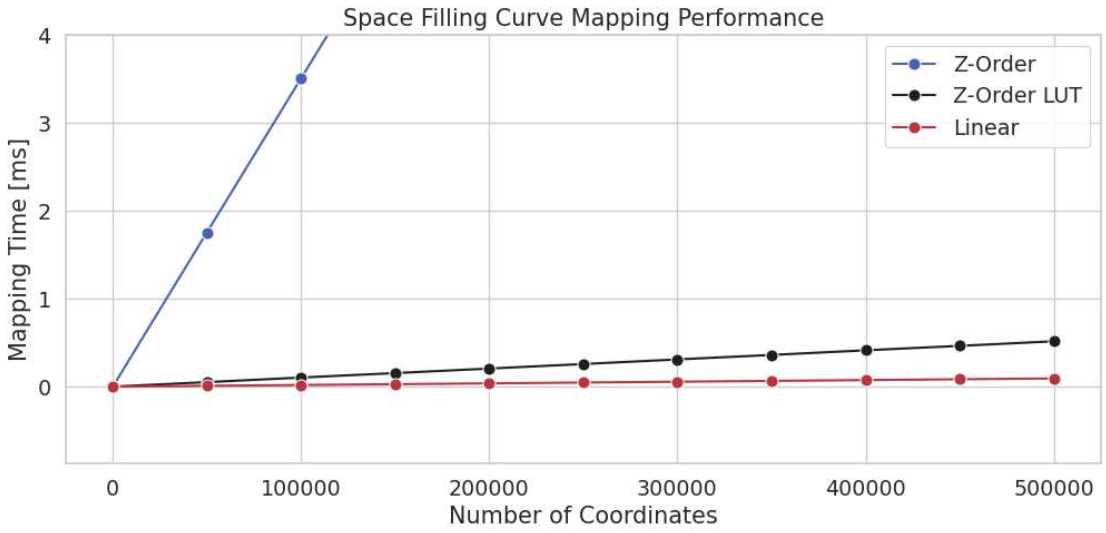

Figure 9.1: Performance comparison of space-filling curve mapping. While the linear mapping is the fastest, the Z-order curve optimization, using a pre-calculated chunk-wise lookup table as explained by Baer [7], proved to be very efficient compared to the standard Z-order mapping implementation.

As illustrated in Figure 9.1, direct Z-order mapping turns out to be more resource-intensive than the optimized lookup table-based approach explained in Section 7.5.2. Although simple linear mapping emerges as the fastest technique, the impact on overall performance is negligible when using the optimized lookup table-based Z-order calculation. It is not typical to use the space-filling curve directly in query processing but rather to optimize data storage and retrieval. This was a design decision in the current implementation to optionally enable the storage engine to optimize data retrieval by using the mapped coordinates for data indexing.

9.2.2 Storage Engines

In this benchmark, I measured the performance of the currently implemented storage engines: the GOB/JSON engine and the memory-mapped index file engine. The BBolt engine was excluded from the evaluation due to its lower performance in initial tests and its slow write times during index creation.

Voxel Grid Indexing

The testing involved retrieving a fixed-size data buffer for a collection of 1D coordinates, such as those from a space-filling curve mapping. Measurements were executed in two scenarios: sequentially using all coordinates and selecting interleaved coordinates from the opposite end of the 1D coordinate range to simulate a worst-case scenario.

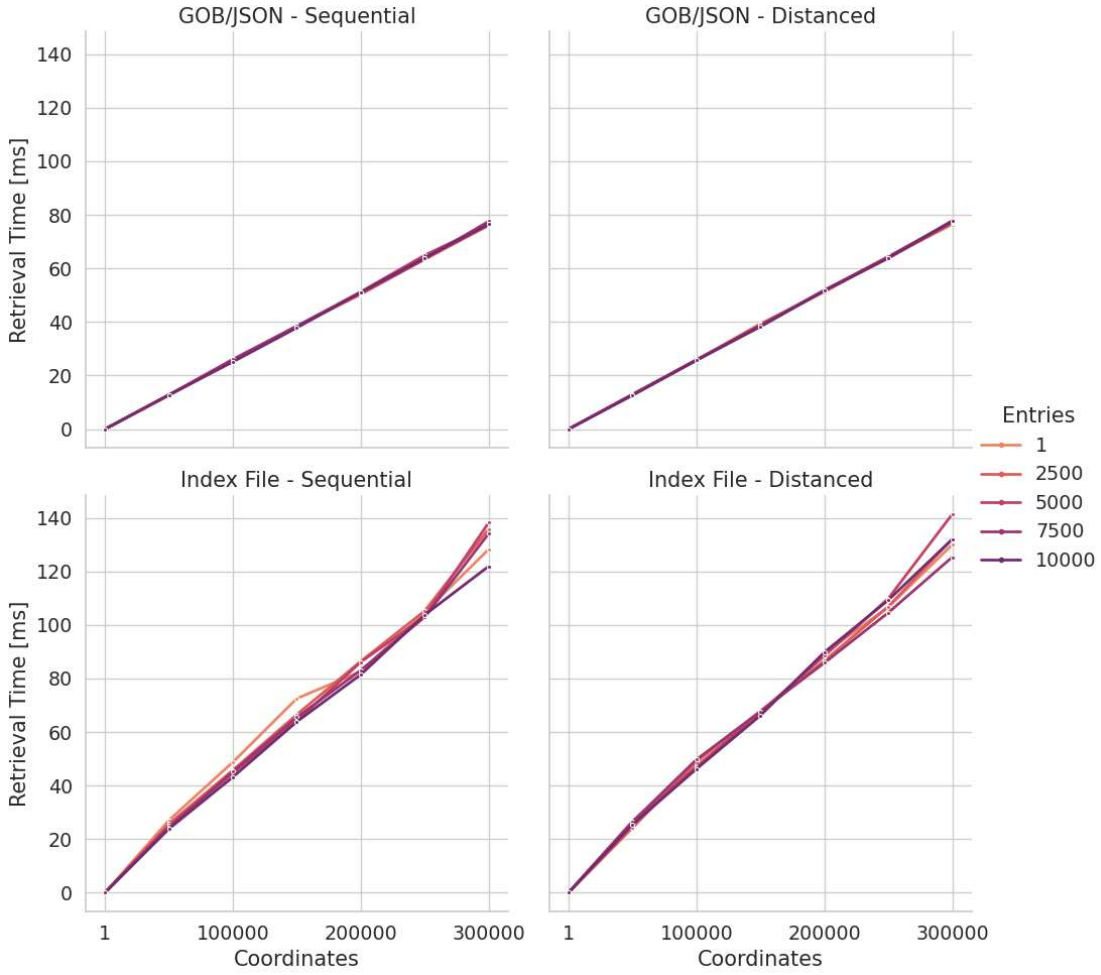

Figure 9.2: A synthetic benchmark of reading voxel data pages of coordinates, comparing the GOB/JSON in-memory storage engine to the memory-mapped index file engine.

As depicted in Figure 9.2, the GOB/JSON engine, functioning as an in-memory storage solution, exhibited the highest speed by requiring the entire index to be loaded into memory. Although the GOB/JSON engine delivered superior performance, it is restricted by available memory, making the memory-mapped index file engine more suitable for handling larger indices or indices with more entries.

The results, as shown in Figure 9.2, indicate that the GOB storage engine outperforms the index file engine by approximately a factor of 1.5, albeit at a higher memory cost. The largest index tested involved 300,000 voxels with 10,000 data entries per voxel, consuming 21 GB of memory for the GOB engine. In contrast, the memory-mapped index file engine required only 22 MB to maintain the indexing structure in memory, which facilitated access to the voxel locations on the disk. The memory mapping of the index file engine consumed about 2 GB during queries. For both engines, query time increased linearly with the number of queried voxels, while the number of entries per voxel had a negligible impact on data retrieval times due to the fast serial reading of the SSD.

Regions and Samples

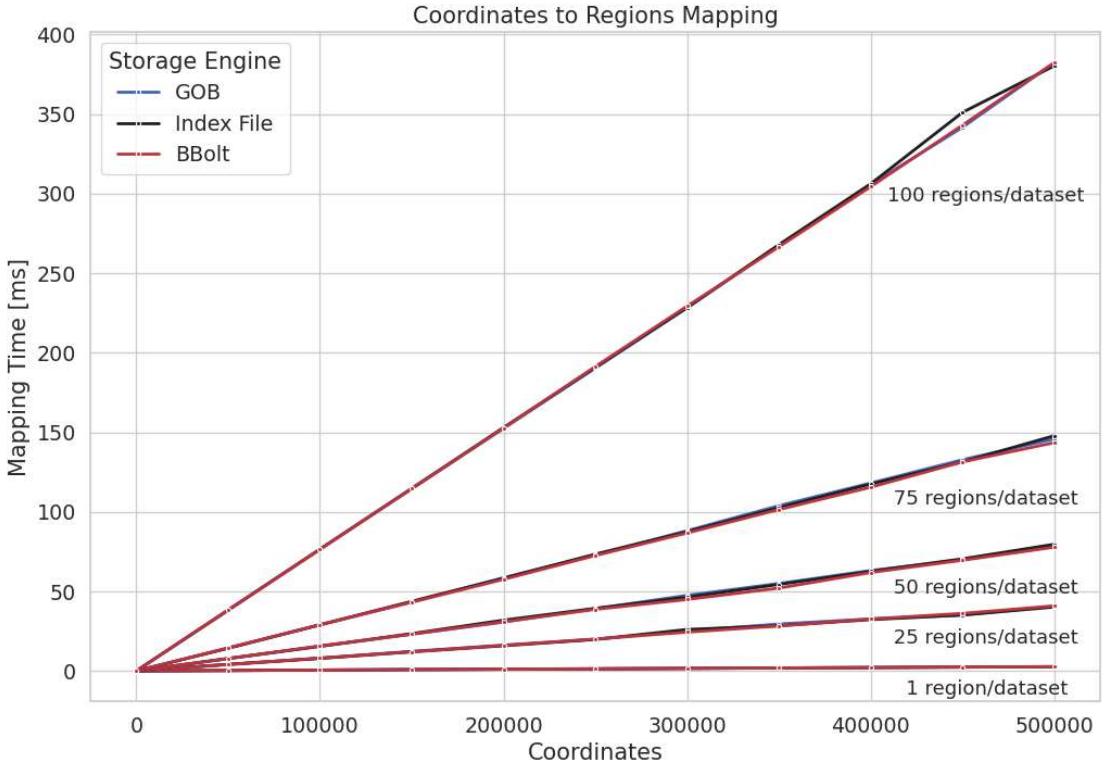

Figure 9.3: Mapping of coordinates to regions, where each coordinate maps to all regions. The measurements include building a unique list of the regions to be read from the set of regions across all queried coordinates.

In the voxel codec index, data retrieval involves a straightforward mapping of coordinates to voxel data pages. In contrast, the region codec index employs a more intricate mapping process that links coordinates to regions, regions to samples, and finally, samples to their respective data. I assessed the performance of the storage engines by measuring the retrieval of regions associated with specific coordinates, then accessing the samples in these regions, and ultimately, retrieving the data buffer of each sample.

The test was conducted using a simple 1D range of coordinates with a fixed number of ten datasets and varying numbers of regions and samples per region across two data layers. By setting all regions at each coordinate, a worst-case scenario was created, which required checking all regions for each coordinate. The measurements were conducted using a synthetic benchmark with a fixed number of datasets and varying numbers of regions and samples per region.

Figure 9.3 illustrates that all three storage engines delivered on-par performance during the initial coordinate mapping. The index file engine and the GOB/JSON engine keep the indexing structures and page tables in memory, whereas the BBolt key-value store uses a memory-mapped storage bucket to manage its indexing structure dynamically. However, the BBolt engine was not included in subsequent data retrieval tests due to its approximately two times slower read performance compared to the index file engine in tests with up to 2000 index items and its slow data page storage times during index creation.

As shown in Figure 9.4, the initial mapping from regions to region samples has little impact on data retrieval performance compared to the data buffer read operations. Reading a second data layer requires approximately the same time as the first layer, with the index file engine currently outperforming the GOB/JSON engine. This advantage likely stems from their differing indexing structures. The index file engine uses an array of pointers to arrays for mapping coordinates to regions, and a map of regions to an array of sample access information. Different from that, the GOB/JSON engine relies on nested maps—from coordinates to regions, regions to samples, and finally samples to data. This way, the index file is able to avoid the hashing process of thousands of coordinates and mapped region ItemIDs (see Section 7.2). This results in more efficient data retrieval.

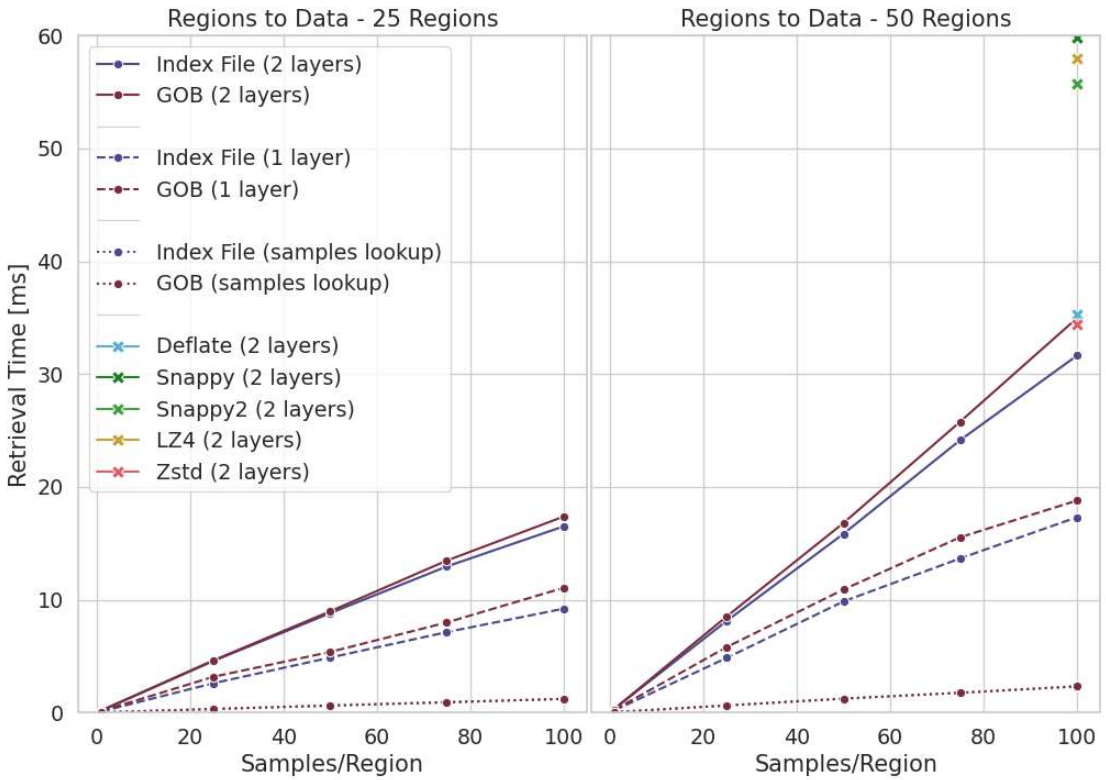

Figure 9.4: Mapping from regions to samples, and further to two layers of different data. Each sample's data needs to be indexed separately, although in sequential order in its respective region. The right side also shows the query time comparing the different inline compression mechanisms on the largest index.

9.2.3 Inline Data Page Compression

Evaluating the effectiveness of compression techniques in this framework is challenging due to the dependency of compression outcomes on the specific nature of the data. Inline compression is employed to compress the data page of a voxel. Each page is compressed individually, and to optimize performance, compression is only applied if the compressed buffer is less than 80% of the size of the uncompressed buffer, minimizing decompression overhead. For this benchmark, I modified this threshold to compress any buffer that was smaller than its uncompressed counterpart, regardless of the 80% limit.

Voxel Payload Compression

The payloads vary by codec: one byte each for the staining codec outlined in Section 8.4.1, the gene-expression-value codec explained in Section 8.7.1, and the distance-field codec described in Section 8.4.2, and three bytes for the structure codec of Section 8.4.3. The benchmark focused on measuring the decompression time and the process of unpacking each entry from an in-memory buffer containing concatenated identifiers and identical

data entries. The results showed that compression was generally ineffective for codecs with a one-byte data entry (staining, distance-field, and gene-expression-value) unless using approximately 200,000+ entries.

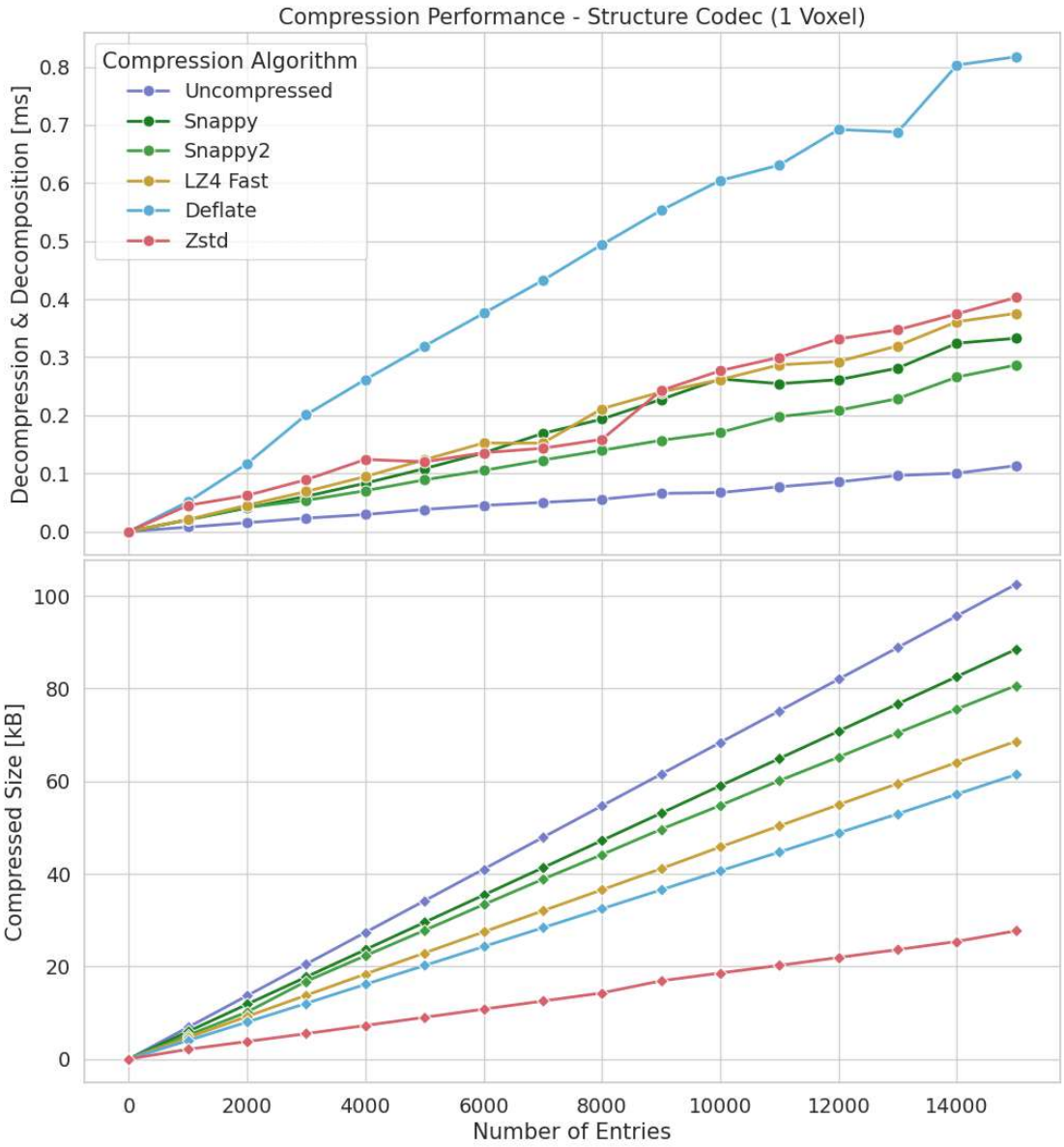

Figure 9.5: Benchmark results of decomposing a compressed voxel data page of a structure index into single index item identifiers and data entries. The results show the decompression time for various compression algorithms.

However, for the three-byte data entry used in the structure codec, compression proved to be more beneficial, as seen in Figure 9.5. Among the compression algorithms eval-

uated, Deflate—being the oldest and offering a moderate compression ratio—was the slowest to decompress. Conversely, Zstd provided the highest compression ratio by a significant margin, whereas Snappy and its reimplementation Snappy2 were the fastest in decompression but offered lower compression ratios.

Region Sample Compression

A voxel codec index is optimized for small, fixed-size byte data entries, whereas the region codec index accommodates data entries of arbitrary sizes, enhancing the potential effectiveness of inline compression, particularly if many similar entries are stored, such as a gene showing no expression, represented as zero.

In my experiments with the synthetic gene-sample-meta index, the first data layer of a sample included typical metadata such as a sample ID, brain region ID (both strings), sex ("M" or "F"), age category ("adult"), and short random strings for phenotype, genotype, and cell type, with two "N/A" fields reserved for annotations. This layer did not achieve significant compression during experiments.

However, the second data layer, containing gene expressions as 32-bit floats, showed potential for effective compression. For benchmarking, ten different 32-bit floating point numbers were used repeatedly as gene expression data. To measure the compression ratio of one gene expression profile as well as possible, only one region with one sample was used, within the fixed ten index items. This resulted in a synthetic gene-sample-meta index with ten data pages storing gene expression values.

Table 9.1 shows the file size of the uncompressed index compared to the one using compression. The results demonstrate that the compression algorithms are becoming more effective in reducing the file size as the number of genes increases in this simplified scenario.

Genes	Size (MB)	Deflate %	LZ4 %	Snappy %	Snappy2 %	Zstd %
1	10.5	100	100	100	100	100
1000	10.7	98.6	98.8	98.9	98.8	98.5
2500	10.9	96.3	96.8	97.1	97.0	96.1
5000	11.4	93.1	93.8	94.3	94.2	92.4
7500	11.8	90.0	91.0	91.7	91.6	88.9
10000	12.3	87.2	88.5	89.4	89.2	85.7

Table 9.1: File size comparison of gene-sample-meta indices utilizing different inline compression mechanisms, demonstrating the effect of compressing synthetic gene expression data compared to uncompressed indices. The indices were created using 10 datasets, each containing 1 region and 1 sample per region. The gene expressions are 32-bit floating point numbers with 10 unique values repeatedly used.

Compression	Regions	Samples/Region	Genes	Index File Size (MB)	%
None	50	100	1000	1626.46	100
Deflate	50	100	1000	1436.85	88.3
LZ4	50	100	1000	1438.20	88.4
Snappy	50	100	1000	1445.86	89.0
Snappy2	50	100	1000	1437.07	88.4
Zstd	50	100	1000	1436.67	88.3

Table 9.2: Size comparison of gene-sample-meta index using small synthetic sample metadata and a synthetic set of gene data for 1000 genes. The indices were created using 10 datasets, 50 regions per index item, 100 samples per region, and 1000 genes per sample (a total of 50,000,000 gene expression values, with 10 unique values repeatedly used). The compression is also applied to the lookups and page tables.

Figure 9.4 also shows the query speed results for both layers across 50 regions with 100 samples each and 1000 voxels per sample. Additionally, the graph displays the time required to read all data from both layers using different compression algorithms with the index file store engine. Notably, indices compressed with Zstd and Deflate performed comparably to the uncompressed index in several tests, with times close to those of the uncompressed index. In contrast, Snappy, Snappy2, and LZ4 consistently required around 55 ms, slower than the 32.5 ms observed with the uncompressed data entry. Tables 9.1 and 9.2 show the results of the experiments in terms of the change in the storage size of the indices.

9.2.4 Result Encoding

In the current implementation, communication with the SPX daemon handling the queries is done via JSON. When this project commenced with Go version 1.10, the JSON serialization in Go's standard library was notably slow. To improve this, the Json-Iterator library [89] was employed, and additionally, custom marshalling with string buffers was implemented to accelerate the JSON serialization of query results. During the course of this project, Go made significant runtime performance improvements, including enhancements to its standard library, particularly in JSON serialization. This led to a reevaluation of JSON encoding performance and the necessity of custom marshalling at the project's end.

The performance evaluation involved a prototype query result comprising an array of integer ItemIDs and float64 values, serialized into JSON using various libraries, including Json-Iterator with custom marshalling. Benchmarks were conducted using Go's benchmark utility over a minimum duration of 20 seconds to ensure stable results.

As seen in Figure 9.6, Go's upgraded standard library encoding (*encoding/json*) made it on par with the Json-Iterator library (*json-iterator*) without the need for any custom string marshalling. This improvement can be attributed to enhancements in reflection

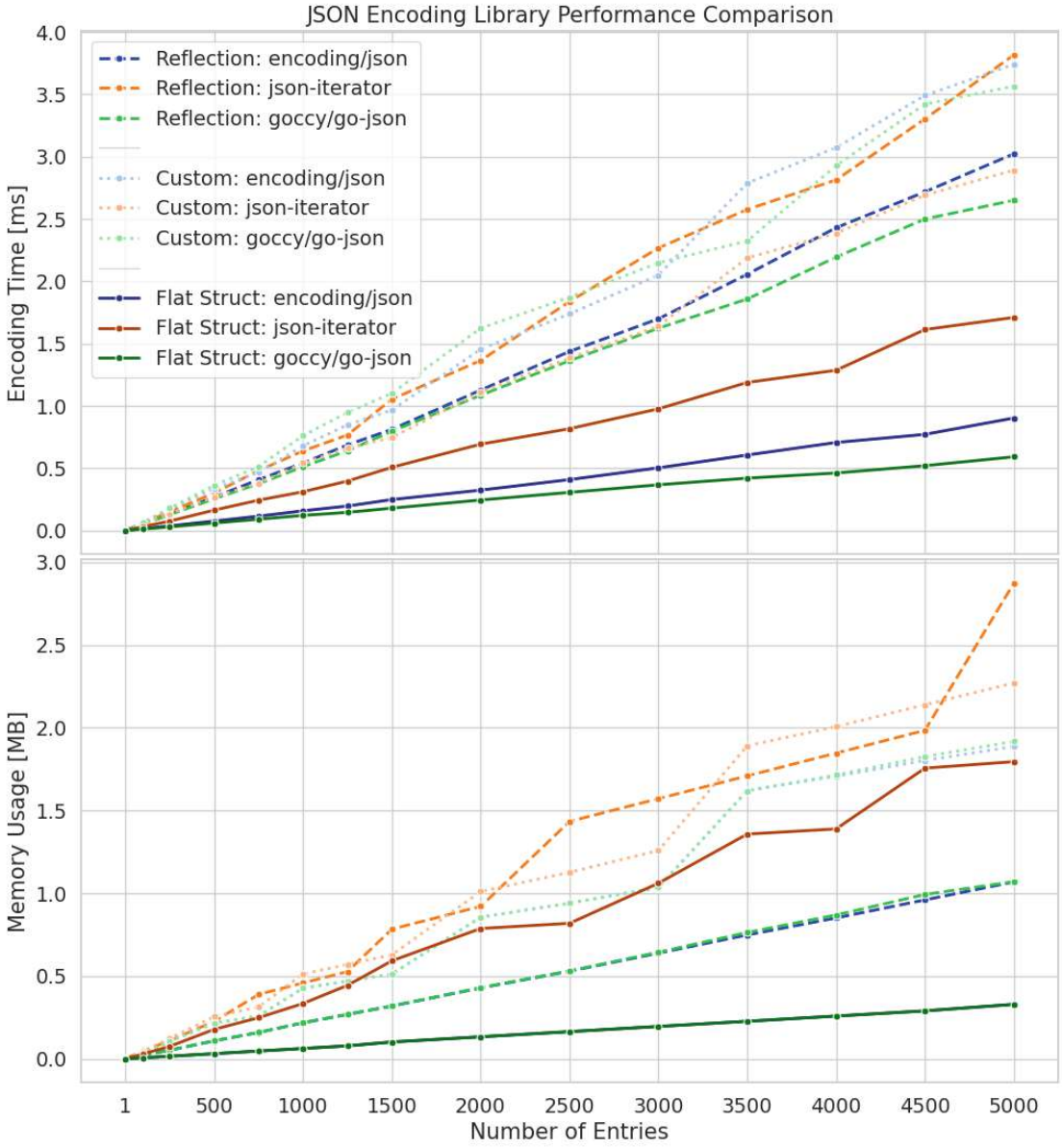

Figure 9.6: A performance comparison of JSON libraries during the encoding of a typical query result, where each entry consists of an integer-based serialized ItemID and a one-byte data entry. The comparison shows the required encoding time and memory usage during the encoding process.

handling, which became faster in recent Go versions, contributing to the performance boost of Go's standard library JSON encoding.
However, significant performance improvements in marshalling could be achieved by rearranging the data from a nested structure into a flat one by embedding the ItemIDs'

attributes directly for serialization. This approach avoids the need for reflection, hence optimizing the performance while directly producing the JSON format required by clients. The main limitation of this method is its reduced flexibility since the native marshalling of Go for the ItemID cannot be used if the ItemID attributes are directly embedded in the result. This embedding required distinct result structures for the two different types of ItemIDs (the string-based one and the integer-based one explained in Section 7.2). Ultimately, a newer library, Go-Json [28] (*goccy/go-json*), demonstrated superior encoding speed with comparable memory usage to the Go standard library encoding. As a result, the Go-Json library is now used throughout SPX, replacing Json-Iterator and the custom result marshalling methods.

9.3 Pipeline Evaluation

While testing individual components of the implementation helps to identify and refine specific settings and discover bottlenecks, certain implementation details cannot be effectively evaluated in isolation. Additionally, the overall performance relies on the interplay between various components, necessitating the evaluation of the entire index pipeline to accurately measure comprehensive performance.

To achieve this, I tested the complete pipeline using a compiled SPX daemon executable and employed the Go unit test framework for scripting and running these tests. The unit test framework in this scenario was only used to create the necessary indices, start the SPX daemon, and perform the actual measurements. This setup accounted for factors such as marshalling and the operating system's network stack, thus providing a detailed assessment of system performance in a controlled environment. The performance measurements were taken from the moment the marshalled query was sent to the SPX daemon to the point when the response was received. The Go unit test framework therefore had no influence on the actual query processing time, as it only measured the time from the query's start to receiving the results.

The query coordinate setup for testing the pipeline was designed to assess system performance under varied operational conditions. Each test was run with a sequential set of coordinates, created by region-growing from a random coordinate until the desired number of coordinates was reached, to evaluate the data ordering of the space-filling curve. Additionally, a random set of coordinates simulated a worst-case scenario by forcing the system to access data entries scattered across the entire indexing structure. Each test scenario was executed 20 times to ensure consistent results.

9.3.1 Voxel Codec Indices

Each test utilized a synthetic dataset of images designed to simulate worst-case properties, meaning that every voxel in the image has a higher than zero expression. This setup diverges from current real-world usage scenarios, where a significant number of voxels, except for those in the central regions of the volume, do not exhibit any expression. To maintain consistency across all tests, each index was constructed with uniform dimensions of 100x100x100 voxels.

Space-Filling Curves

The fundamental aspect of the original spatial index idea was its reliance on space-filling curves to order data. Originally, Schulze's Java implementation utilized the Hilbert curve, while the new Go implementation adopts the Z-order curve and includes a linear mapping option for comparative analysis. To assess the impact of these space-filling curves on performance, I compared the query times of a linear mapping with those of the Z-order curve. The index was constructed using the staining codec, representing 1,500 synthetic staining images, with each coordinate holding values from all images. Queries were performed across increasing numbers of coordinates (300,000 to 500,000).

Figure 9.7 shows the median query times and the 95% confidence intervals, performance differences on a modern high-performance SSD are small when switching between different data areas, especially if the SSD's cache is large enough to hold the data. The Z-order curve's primary advantage is its ability to optimize data access by maximizing block-wise reading and caching. Sequential reading of larger data blocks is faster than a random access pattern, although not significantly, which is due to the high-speed SSD's ability to read data in parallel from different locations.

The Z-order curve provides only a small performance gain, if at all, on a modern high-performance SSD with hundreds of GB as cache, but comes with the cost of the computational overhead of mapping the queried coordinates. In the current implementation, the storage engine uses the mapped coordinates to optimize storage access. The Hilbert curve, while offering better locality preservation, requires a more complex calculation process and would therefore be slower than the optimized Z-order curve mapping in this context. In environments using magnetic spinning disk drives, the Z-order curve could potentially offer more significant advantages due to effective read-ahead caching and the higher cost of random access. However, the prevalence and affordability of high-speed SSDs make this a less relevant scenario today. In the used memory-mapped index file storage engine, forward-read-flow synchronization, described in Section 6.4.3, is enabled, which organizes the requested coordinates according to the space-filling curve to reduce the inter-file jumps and guarantees that the data is read in an as sequential as possible order.

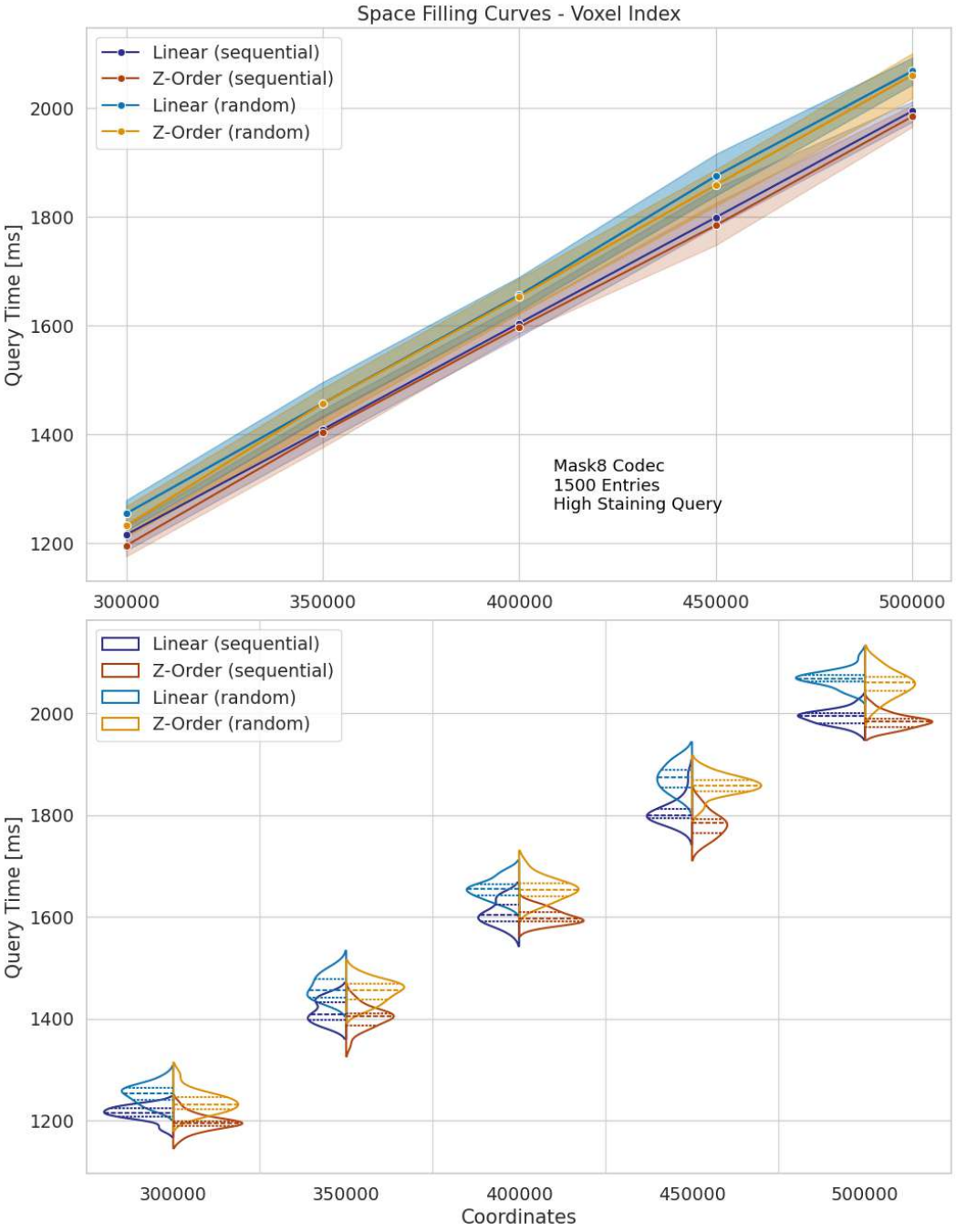

Figure 9.7: Comparison of query times using the optimized Z-order curve implementation and a simple linear mapping with either a random set of coordinates or a sequential set. Each test was run 20 times. The lines represent the medians, while the shaded bands indicate the 95% confidence interval. The violin plots show the distribution of the query times.

Multithreading

A key enhancement in the new architecture is its robust multithreading support, which significantly optimizes the efficiency of the index. This feature allows for the parallel unpacking and decoding of index items and data entries and supports concurrent and/or parallel processing in codecs, provided that the codec's design accommodates this. To benchmark the multithreading capabilities against the previous single-threaded architecture, I restricted the number of Go-routines to one in the new setup. In the multithreaded version, it preallocates Go-routines ten times the number of supported CPU threads since Go-routines are green threads. Despite the limitation to one routine, the improved processing in the codecs, along with an active separate read routine and the data streaming pattern, ensures that the performance of this single-routine setup still surpasses that of the older architecture. Measurements of that setup can be found in Schulze et al.'s report [76].

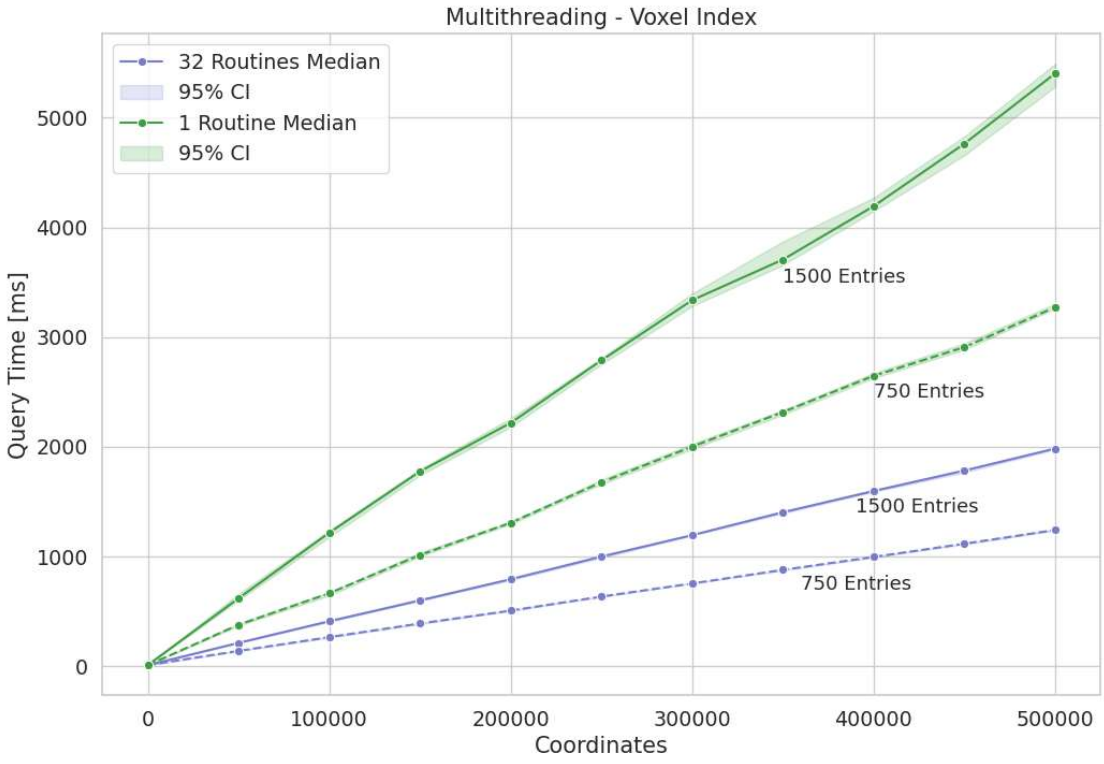

Figure 9.8: Simulation of the single-threaded architecture of the prototype at the project start compared to the multithreaded architecture of the current implementation, using the high-staining query on a staining index.

For the test, a staining index with 1,500 and 750 entries was employed, utilizing the memory-mapped index-file storage engine, performing high-staining queries without the use of inline compression. The query coordinates were sequentially chosen, utilizing a

Z-order space-filling curve. The staining codec's high-staining query result calculation, explained in Section 8.4.1, involves a simple bit count ratio and requires only small computational effort. As seen in Figure 9.8, the multithreaded architecture demonstrates a distinct performance advantage over the original single-threaded prototype.

Storage Engines

In this evaluation, I compared the performance of the in-memory GOB/JSON engine, explained in Section 7.6.1, with the memory-mapped index file storage engine of Section 7.6.4. The test used a staining index with 1,500 image entries and employed the high-staining query, selected for its small computational demands. 1,500 index items were the closest rounded maximum that could be saved and restored by the GOB/JSON engine within the constraints of 32 GB of memory. The BBolt storage engine was excluded from this test due to previous results indicating its slower performance relative to both the index file and GOB/JSON engines, as well as the excessively long index creation time.

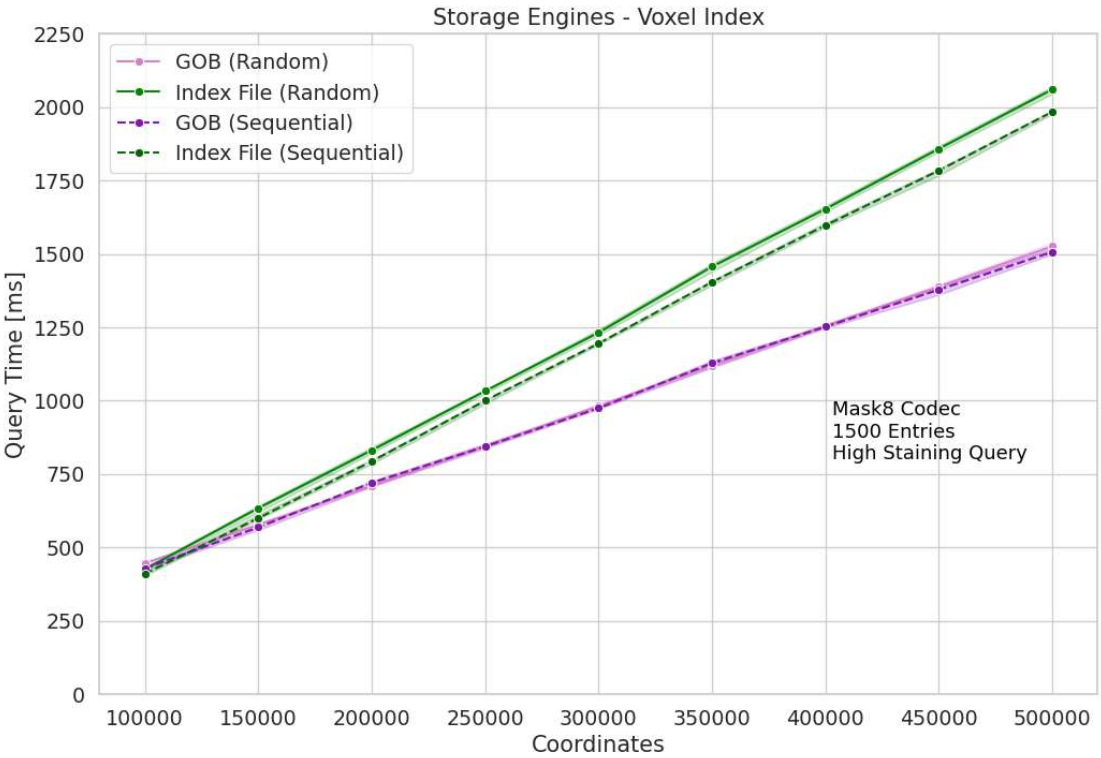

Figure 9.9: Performance comparison of the GOB/JSON storage engine and the index file engine. Measurements were taken with both randomly distributed coordinates and sequentially ordered ones. The range band represents the 95 % confidence interval.

As shown in Figure 9.9, the GOB/JSON engine demonstrated a performance advantage

in data retrieval. Querying 500,000 coordinates with 1,500 index items in an index with a side length of 100 voxels consumed approximately 28 GB of memory (including the index data), compared to about 4.5 GB used by the memory-mapped index file storage engine during querying. Consequently, while the GOB/JSON engine is not suitable for large indices or indices with a multitude of entries, it may still be preferable in use cases involving smaller indices that require higher data access performance.

Inline Data Page Compression

To show the scaling effect of applying compression in the index, I conducted measurements across several staining and structure indices with an increasing number of index items, using the memory-mapped index file storage engine for all tests. The measurements of other implemented codecs, such as the gene-expression-value codec or the distance-field codec, are similar to the measurements of the staining codec using the high-staining query due to their similar data entries and comparable computational requirements in terms of processing them.

In testing various compression libraries on a synthetic *staining index*, Zstd, Snappy2, and Deflate yielded small to no compression. Only Snappy and LZ4 achieved a significant compression ratio, with LZ4 performing the fastest. These measurements are illustrated in Figure 9.10, and Tables 9.3 and 9.4 provide an overview of the change in storage sizes with different compression mechanisms applied to synthetic staining/distance-field and structure indices. In the *structure index*, all tested compression mechanisms demonstrated significant effectiveness in terms of index storage size reduction, as seen in Figure 9.10. Especially utilizing Zstd, the structure codec's three-byte data entry achieved a compression ratio of 85 % while still maintaining an acceptable query time.

Compression	Storage Size (GB)	Storage Size %
Uncompressed	7.1	100
LZ4	5.7	80
Snappy/Snappy2	5.7	80

Table 9.3: Overview of storage size and memory usage of a query when the inline compression is active on a staining index. The results for a distance-field index with applied compression are similar due to the same data entry size.

Nevertheless, the inline compression lowered the query performance in all cases. Reading the uncompressed, larger amount of data from a high-performance SSD is faster than reading the compressed data and decompressing it. The performance gain from reading less data is not enough to compensate for the decompression overhead.

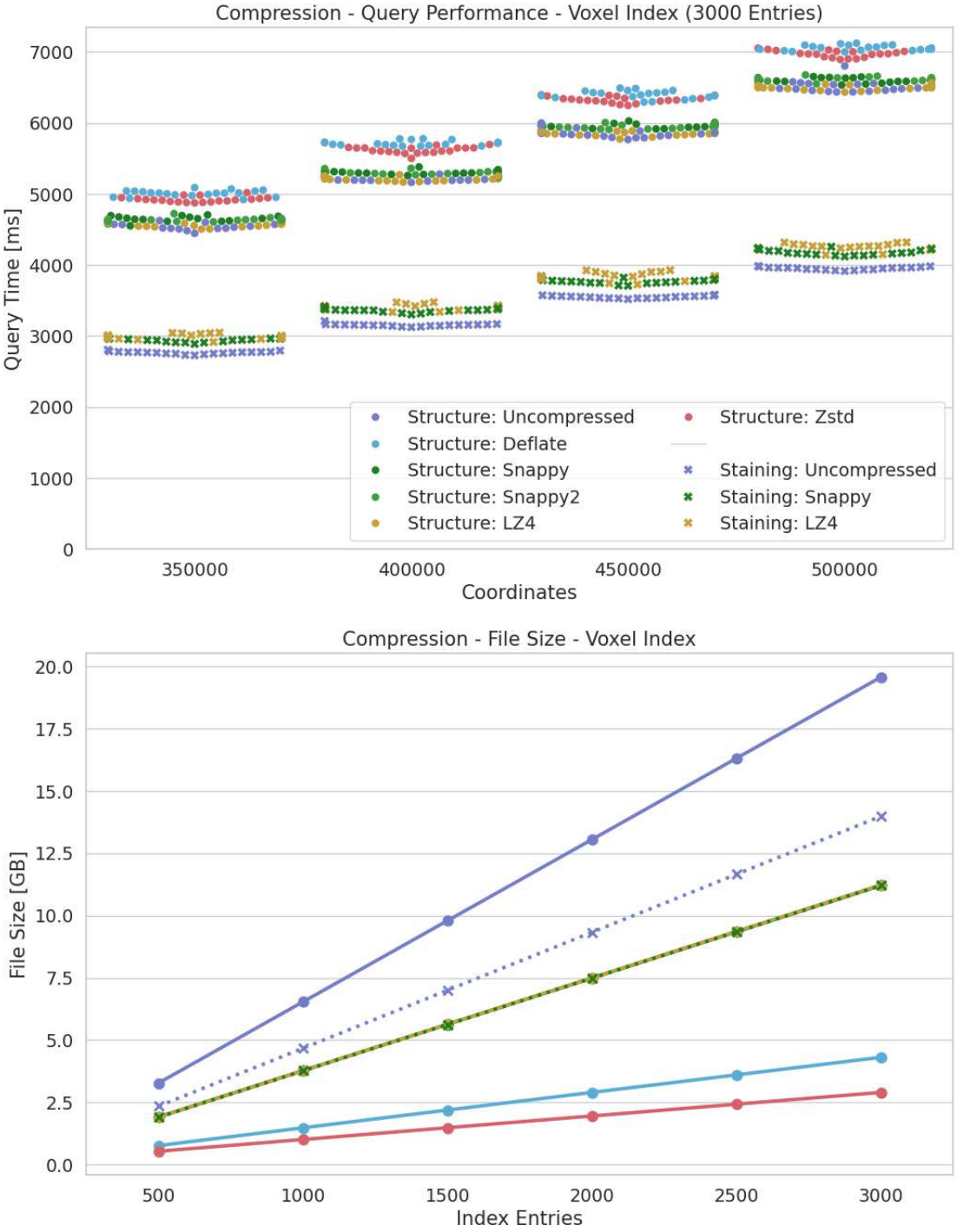

Figure 9.10: Query time and file size benchmark results of applying inline compression on a structure and a staining index.

Compression	Index Storage Size (GB)	Relative Storage Size %
Uncompressed	19.9	100
Snappy/Snappy2	12	60
LZ4	12	60
Deflate	4.4	22
ZStd	2.9	15

Table 9.4: Inline compression used in a test structure codec index.

9.3.2 Region Codec Indices

Each region codec index performance test utilized a synthetic set of regions and region samples. The mapping of coordinates to regions had worst-case properties, where each coordinate had all regions assigned. This setup required evaluating all region entries at each coordinate to generate a unique set of regions — a scenario that typically does not occur in real-world applications. Again, the indices have dimensions of 100x100x100 voxels. A static number of ten datasets was assumed for simplicity, with varying numbers of regions and region samples, where each index item has the same number of regions and region samples.

Space-Filling Curves

In the tests for the region codec index, utilizing the locality-preserving Z-order curve did not yield a measurable performance improvement over linear mapping, as detailed in Figure 9.11. For the region codec index, the effect of locality-preserving data order diminishes because regions span several voxels, reducing the benefits associated with the locality enhancements that the Z-order curve provides.

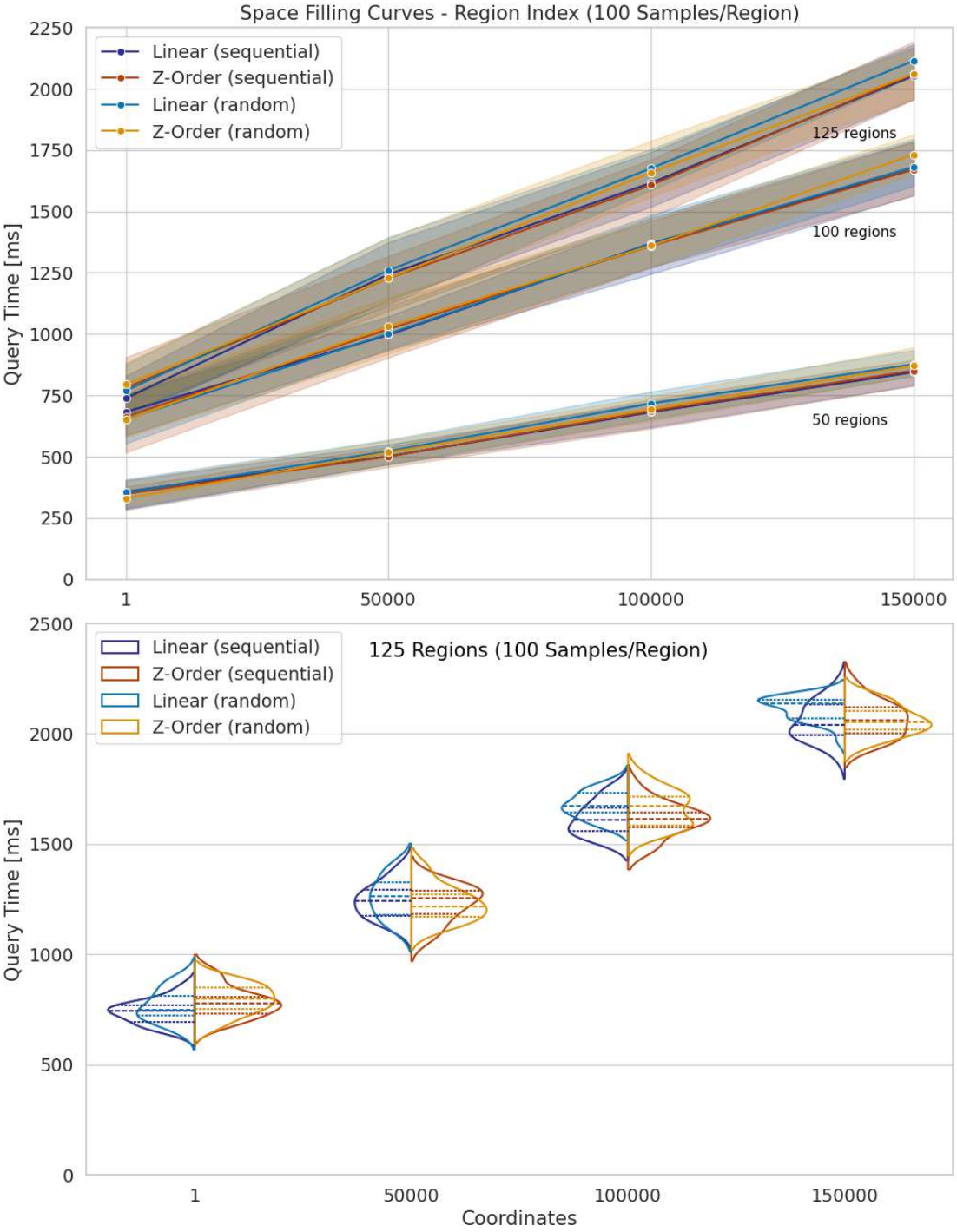

Figure 9.11: Benchmark results of utilizing the optimized Z-order curve implementation and the linear mapping for a region codec index.

Multithreading

Figure 9.12 illustrates the effects of multithreading on query performance in the framework. The tests involved ten index items with varying numbers of regions (1, 25, 50) per index item and sample counts per region (1, 25, 50, 75, 100), ranging from a minimum of 10 to a maximum of 50,000 samples, each containing 1000 gene expressions. Since all regions were evaluated at each coordinate, all regions were included in the evaluation.

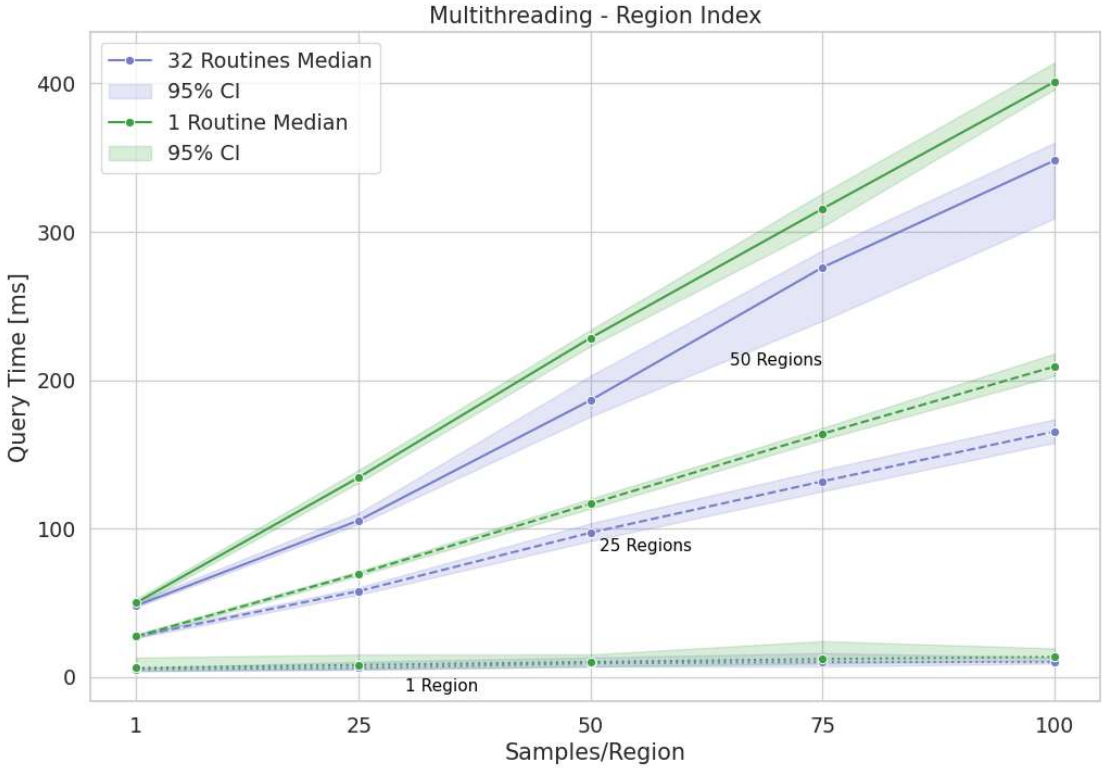

Figure 9.12: The usage of multiple Go-routines for the decoding of sample data entries in a region codec index, which is based on the *gene-sample-meta codec*, shows diminishing returns after all Go-routines are saturated with data and are waiting for their turn to process them. This occurs due to the expensive result aggregation in the codec.

In the case of the *gene-sample-meta codec*, multithreading is only utilized in the decoding of the region sample's data entries, as the codec's result aggregation in its current implementation is not optimized for multithreading or efficiency. The default dataset categorization is used. The primary purpose of the codec is the aggregation of specific gene expressions according to certain queried metadata attributes (categories) and values (categorization values), as explained in the description of BrainTrawler 8.5 and BrainTrawler Lite 8.6. This is currently done using a complex nested map of maps structure, spanning the number of categorization sample properties chosen by the user.

Each level of nesting requires a hashing operation of the categorical metadata value of a sample to find and access the next level, or to create a new entry in the current category map. The observed performance gain in Figure 9.12 is therefore only given up to the point where all Go-routines are saturated with data and waiting for their turn to process that data in the result aggregation.

Storage Engines

Since the result aggregation is the most time-consuming part of the gene-sample-meta codec, the storage engine has currently limited influence on the performance of the region codec index. This is visible in Figure 9.12, where the change in slope occurs once the worker Go-routines experience delays in their job processing due to result aggregation by the codec. If data retrieval were the bottleneck, SPX would never reach the situation were all Go-routines are waiting for their turn to process the data, and the line would have a constant slope.

Inline Sample Data Page Compression

In this measurement, I used 125 regions with 100 samples in all of the ten datasets, and each sample contained 1000 gene expression values. Again, only one coordinate was used to minimize the influence of coordinate mapping and to simulate direct region queries. Similarly to the choice of the storage engine, the inline compression currently has a marginal influence on the query performance of the gene-sample-meta codec, as the codec's result aggregation is the costly part. In this case, using inline compression can save storage memory without affecting the query time, except when LZ4 compression is used. Interestingly, the usage of Zstd seems to stabilize the query time, as seen in Figure 9.13. Table 9.5 shows a relative file size comparison of the compressed test gene-sample-meta index.

Compression	Index Storage Size (GB)	Relative Storage Size %
Uncompressed	4.88	100
Snappy	4.40	90.0
Snappy2	4.37	89.5
LZ4	4.38	89.8
Deflate	4.37	89.5
ZStd	4.37	89.5

Table 9.5: Relative file size reduction of applying inline compression to a synthetic gene-sample-meta codec index. The index was created using 10 index items, 125 regions per index item, 100 samples per region, and 1000 gene expressions per sample.

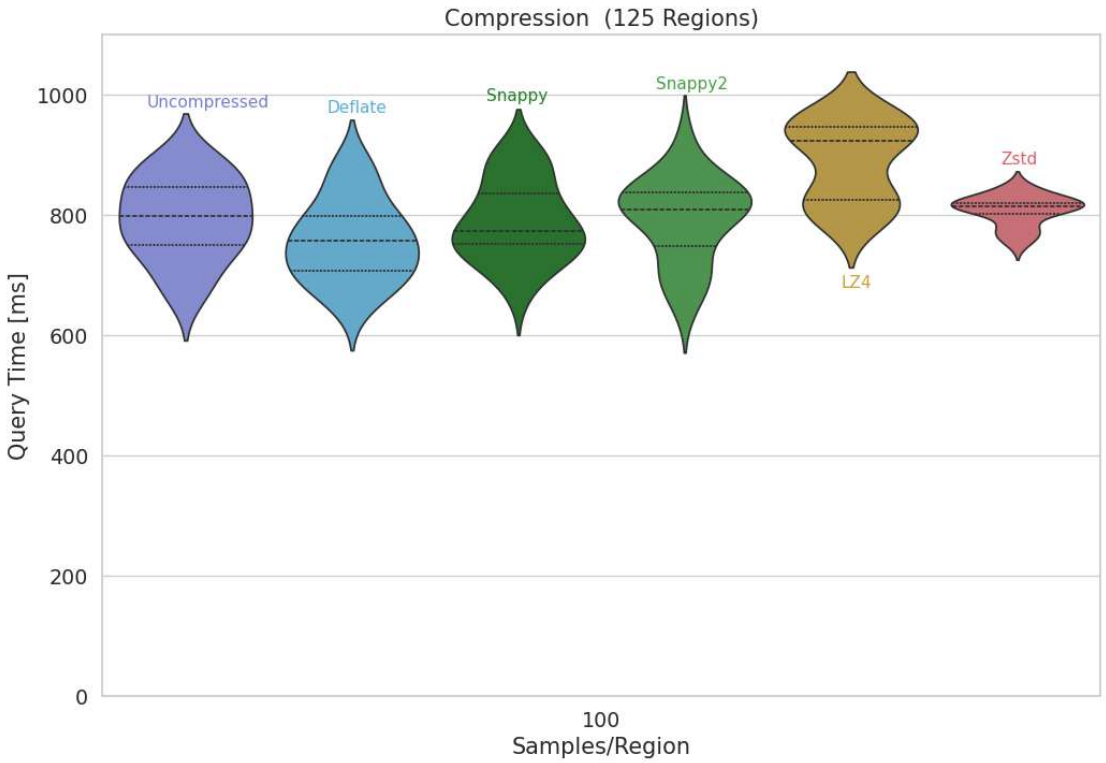

Figure 9.13: Query performance measurements for a region codec index based on the gene-sample-meta codec with 10 datasets, 125 regions, 100 samples per region, and 1000 gene expressions per sample.

9.4 Result Summary

The evaluation of the spatial index's performance across various components and configurations provides a comprehensive overview of the index framework's capabilities and limitations. The results demonstrate the index's efficiency in handling large-scale data queries, with the memory-mapped index file storage engine proving particularly effective in managing large indices with small memory usage. The space-filling curve mapping, while beneficial in optimizing data access, has only a minor impact on performance when using a modern high-speed SSD. The computational overhead of calculating these mappings can be considerable, suggesting that bypassing the mapping calculation could be beneficial if the storage engine directly supports coordinates for data page access while querying.

The effectiveness of inline compression depends significantly on the index data and the processing speed required to compute query results. While compression can effectively reduce storage requirements and enhance performance if codec's data processing is the limiting factor, it may adversely affect query performance if data retrieval becomes a bottleneck. Selecting the right compression algorithm is crucial, with Zstd and Snappy

providing the best balance between performance and compression ratio, favoring either speed or compression ratio respectively. Nevertheless, the performance gain from reduced data size to read is not enough to compensate for the decompression overhead in the currently implemented storage engines, especially when using high-speed SSDs, which over time will become even faster, larger, and more affordable. Still, the inline compression can be used in combination with an in-memory storage engine to keep a larger amount of indexed data in memory.

The multithreading capabilities of the index framework can significantly enhance query performance if the codec's data processing allows for parallelization or if the result processing requires only little computational effort. Go-routines are lightweight and can scale well with multiple CPU threads. However, if the bottleneck lies in the codec's result aggregation, the performance gain diminishes once all routines are waiting to process data.

The evaluation of the voxel codecs demonstrated that the system is well-suited for high-throughput processing of large-scale data queries. The region codec index also showed promising results until the result processing of the gene-sample-meta codec became the bottleneck due to its complex result aggregation.

In the case of multiple indices on the same storage media running multiple queries, data retrieval during concurrent queries can become a significant bottleneck. This challenge can be alleviated by distributing indices across multiple storage devices or implementing a query distribution and balancing mechanism. The current system employs a locking mechanism where only one index can be queried at a time. For multiple queries on the same index and the same query coordinates but with different parameters, the multi-query implementation outlined in Section 6.4.5 can be used. This approach avoids cache misses by maintaining the forward-read direction of the storage engine through the forward-read-flow synchronization discussed in Section 6.4.3, and prevents the loss of allocated decompression caches when switching between different compression algorithms. While the core of SPX, including the space-filling curve implementation and available storage engines, is well optimized for high-throughput data retrieval, the system's performance can be further enhanced by optimizing the codec's result aggregation, especially to fully utilize the multithreading capabilities.

CHAPTER 10

Conclusion

In this thesis, I presented the architecture and implementation of a flexible spatial indexing framework, named SPX, designed to optimize the querying of large-scale spatial grid-based data. The development, deployment, and integration of SPX into research applications such as BrainBaseWeb, BrainTrawler, and BrainTrawler Lite have demonstrated its advancements in spatial data handling, adaptability to various data types, extensibility regarding indexing methods and data storage, and efficient query processing.

By defining two basic types of spatial index data—voxel grid data and region data—and respective codec interfaces for creating, preprocessing, encoding, and processing queried index data, the SPX framework has proven to be highly flexible. It meets the diverse requirements of the BrainBaseWeb, BrainTrawler, and BrainTrawler Lite applications. Several voxel codecs, such as staining, distance-field, structure, and gene-expression-value codecs, were implemented within the SPX framework, along with a region codec called the gene-sample-meta codec. These implementations have enabled SPX to address different research questions based on varying spatial data types.

10.1 Key Findings

The key findings by evaluating the SPX framework in terms of performance and scalability in its current use cases are:

Data Retrieval Performance

A dense, memory-mapped file-based storage engine with in-memory lookup structures provides a scalable and performant data storage solution suitable for large datasets. The use of a modern high-speed SSD ensures that data access and retrieval times are fast enough to keep the query engine saturated with data. In edge cases where rapid data

access is crucial, an in-memory storage engine can be used if sufficient memory is available to store the entire index.

Efficiency and Scalability

Utilizing a producer-(multi-)consumer pattern combined with a job-dispatch system, and pools for Go-routines and buffers, the system has shown efficiency and scalability in processing spatial data in parallel. The multithreaded query engine efficiently utilizes available CPU cores to decode spatial data entries in parallel, thereby timely supplying the codecs with data for processing. This framework has proven capable of handling large datasets and queries and can scale with the number of CPU cores available, provided that the codec's data processing is fast enough.

Codec Performance Dependency

Querying performance is largely dependent on the codec's data processing capabilities. On modern high-speed SSDs, data access and retrieval times are sufficiently fast, allowing the query engine to remain saturated with data. However, current performance is constrained by the fact that the codecs do not utilize the framework's multithreaded data processing. Particularly in scenarios where data processing is the bottleneck, the query engine's performance improvements from employing its multi-routine architecture diminish.

Space-Filling Curves

Using space-filling curves to order data in the index proved beneficial, albeit less than expected and only to a limited extent on modern high-speed SSDs. This is due to the fact that SSDs can read in parallel from multiple places and are not dependent on the positioning of a read head on a spinning disk. Using a space-filling curve can still be beneficial in scenarios where the data is stored on slower storage devices, in a distributed storage system, and to improve the cache hit rate.

While the Z-order curve is primarily used, the SPX framework can accommodate other curves as well. The framework provides an easy way to experiment with different curve types to optimize data retrieval performance and adjust the indexing strategy to enhance caching on a case-by-case basis. However, in its current implementation, the space-filling curve mapping is performed for each coordinate during queries, allowing the storage engine to optimize data retrieval by using the curve's coordinate instead of the 3D one. If a space-filling curve's coordinate mapping is computationally expensive, it is recommended to pre-calculate the mapping for the entire index or to make the mapping calculation optional.

Inline Compression Algorithms

The SPX framework offers a variety of compression algorithms, and different algorithms have shown varying results in terms of performance and compression ratio. The choice of

compression algorithm should be based on the data characteristics, and its effectiveness should be tested on a case-by-case basis to achieve the desired trade-off between performance, compression ratio, and additional memory and CPU usage during decompression. In the current tests, performance was mainly limited by the codec's result aggregation, so the idea of utilizing transparent inline compression to read less data from the data storage in I/O-bound scenarios to improve performance could not be proven, as I/O operations on the test hardware turned out to be sufficiently fast. However, even in data processing-bound index scenarios, compression can lower storage memory requirements, thus enabling in-memory indices to contain more items, at the cost of additional CPU usage for decompression.

Data Layering

Using multiple data layers proved beneficial for the gene-sample-meta codec's use cases, storing metadata and a large amount of gene expression data in the same index. The hierarchical query mechanism allows for reading the extensive gene expression data of an entry only if the metadata matches the query criteria. This approach significantly reduces the amount of data that needs to be read from storage, thereby improving query performance and reducing the memory requirements for the codec's result aggregation. It also provides voxel codecs with more options to store several types of data in the same index, although this has not been utilized in the current codecs.

In conclusion, the architecture and implementation of the SPX framework provide a solid foundation for ongoing and future research on spatial grid data, highlighting its potential for extensive scalability and adaptability across varied research and practical applications.

10.2 Future Extensions and Optimizations

This section outlines potential optimizations, enhancements, and extensions to the existing framework, aiming to boost its performance, scalability, and utility.

10.2.1 Possible SPX Optimizations

While the SPX framework is already highly optimized, there are several areas where further improvements can be realized:

Bounding Box Indexing

Currently, the SPX framework indexes the entire coordinate space, which raises the memory requirements for the in-memory lookup structures and can lead to unnecessarily large indices, especially if the data occupies only a central portion of this space. Implementing a bounding box approach to only index the relevant data space could lead to a reduction in memory and storage requirements. The bounding box properties could be stored in the

header and calculated by the codecs during the indexing process. Requested coordinates outside of the bounding box would not need to be considered.

Space-Filling Curve Mapping

The current implementation involves recalculating the space-filling curve mapping for each coordinate during queries. The Z-order curve using the lookup table-based optimization, as well as the linear mapping, is inexpensive to calculate and can be done for each coordinate. However, depending on the space-filling curve, calculating the mapping can be computationally expensive. This could be solved in two ways:

- Pre-calculate the mapping for the entire index for costly space-filling curves.
- Make the mapping calculation optional, allowing the storage engine to decide if it is required for fetching the data.

Possible SPX Extensions

Several extensions can be considered to broaden the SPX framework's applicabilities and enhance its functionality:

LRU Page Caching

In the current approach, each query is handled separately, with index pages loaded, processed, and discarded. One option to improve performance would be to use a Last-Recently-Used (LRU) type cache to keep frequently requested pages in memory. The in-memory lookup structures can be extended to keep track of the cached voxel or region sample data pages.

Voxel Codec Indices with Varying Payloads

The current implementation requires the same entry payload size across each data layer of a voxel codec index. This could be easily extended to support different payloads for each layer, allowing for more complex data structures.

Index Pyramid

Adding support for multiple resolutions of the data grid could foster more efficient querying in specific scenarios by utilizing pre-calculated data for lower voxel resolutions. If the query region spans the complete area covered by the pre-calculated data, only areas not completely covered by the query area would need to be fetched in the highest resolution. This approach is similar to that used by Herzberger et al. [35], in which they use an octree to determine data bricks in different resolutions required for rendering.

Time-Series Data

Utilizing the data layer concept to support time series data by employing a predictive-frames concept, which encodes only the differences relative to the last layer, similar to video compression techniques. This helps to reduce the required data storage and increases data throughput.

Complex Multi-Queries

Another possible extension of SPX would be to enable the combination of query results, such as the intersection of queries, the union of queries, or the difference between two queries, by extending the multi-query approach.

New Queries - New Codecs

The SPX framework's current codecs are driven by the research questions addressed in the deployment environments and applications. New research questions will require new codec implementations or an extension of the available codecs with new query ways and options. The SPX framework offers an easy way to integrate ideas described in the literature. One example would be the integration of the concept described by Weissenböck et al. [88], where they use the space-filling curve mapping to create comparative line and heatmap visualizations based on a scaling of the space-filling curve line mapping with a variance importance function. Data for this could stem from an already existing codec by adding the required queries to it or by creating a new codec, depending on the type of data the query should operate on.

New Storage Engines

Using the storage engine interface, other data sources like databases or real-time data streams can be integrated into the SPX framework, providing new data source options. This could be beneficial for real-time data processing or if the data is too large to be stored on a single machine.

Codec Optimization and Extensions

The current codecs' methods for processing query data to aggregate results are thread-safe but do not leverage the SPX framework's multi-routine capabilities. Implementing the available result aggregation methods to utilize these capabilities could significantly enhance the processing of retrieved spatial data in parallel. Given that SPX's benchmarks identify codec data processing as a critical bottleneck, optimizing the codecs to process data concurrently could lead to considerable performance improvements. This optimization might involve employing a *fork-join pattern* [51] to process subsets of the spatial data simultaneously or utilizing a *map-reduce pattern* [51] to compute intermediate results in parallel and then combine them.

The Gene-Sample-Meta Codec

Each aggregation category utilizes a hash map, which becomes increasingly nested with each requested category. An opportunity for optimization exists in replacing this structure with a more efficient data structure designed specifically for handling aggregation categories.

Staining Codec, Structure Codec, and Distance-Field Codec

All three provide queries operating on similarity towards a given reference image or segmentation. In both cases, the queries require the reference image or segmentation to be part of the indexed data. A helpful extension would be to optionally provide the comparator data for the query area, enabling the utilization of the queries for images and structures not included in the indexed datasets.

Overview of Generative AI Tools Used

ChatGPT 4 - August, 2024
The free version of ChatGPT 4 was used to check grammar, spelling, and punctuation in cases I was unsure. Further, it helped in a few sections to improve the flow, when I was unable to find proper wording. It was not used to contribute content in any way.

List of Figures

List of Listings

Bibliography

[1] T. Akenine-Möller and E. Haines. *Real-time rendering*. CRC Press, 2002.

[2] M.A. Akram et al. *An open repository for single-cell reconstructions of the brain forest*. In: Scientific Data 5.1 (2018), p. 180006. ISSN: 2052-4463. DOI: 10.1038/sdata.2018.6.

[3] J. Alber and R. Niedermeier. *On Multidimensional Curves with Hilbert Property*. In: Theory Comput. Syst. 33 (Aug. 2000), pp. 295–312. DOI: 10.1007/s002240010003.

[4] K. Amunts et al. *The BigBrain Atlas*. In: Nature 590.7845 (2016), pp. 541–547.

[5] J.D. Armstrong et al. *Flybrain: An Online Atlas and Database of the Drosophila Nervous System*. In: Neuron 15 (1995), pp. 17–20.

[6] M. Ashburner et al. *Gene Ontology: tool for the unification of biology*. In: Nature Genetics 25.1 (2000), pp. 25–29. DOI: 10.1038/75556.

[7] J. Baert. *Morton Encoding/Decoding Through Bit Interleaving: Implementations*. Oct. 7, 2013. URL: https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/ (visited on 01/10/2022).

[8] N. Beckmann et al. *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles*. In: SIGMOD Rec. 19.2 (May 1990), pp. 322–331. ISSN: 0163-5808. DOI: 10.1145/93605.98741.

[9] S. Berchtold, D.A. Keim, and H.P. Kriegel. *The X-tree: An Index Structure for High-Dimensional Data*. In: Proceedings of the 22nd VLDB Conference. Mumbai, India, 1996, pp. 28–39.

[10] A. H. Brand and N. Perrimon. *Targeted Gene Expression as a Means of Altering Cell Fates and Generating Dominant Phenotypes*. In: Development (Cambridge, England) 118.2 (1993), pp. 401–415. DOI: 10.1242/dev.118.2.401.

[11] S. Bruckner and E. Gröller. *Instant Volume Visualization using Maximum Intensity Difference Accumulation*. In: Computer Graphics Forum 28.3 (June 2009), pp. 775–782. ISSN: 0167-7055.

[12] S. Bruckner et al. *BrainGazer - Visual Queries for Neurobiology Research*. In: IEEE Transactions on Visualization and Computer Graphics 15 (Nov. 2009), pp. 1497–504. DOI: 10.1109/TVCG.2009.121.

[13] Janelia Research Campus. *FlyEM*. URL: https://www.janelia.org/project-team/flyem (visited on 03/04/2023).

[14] M. Chalfie et al. *Green Fluorescent pProtein as a Marker for Gene Expression*. In: Science 263.5148 (1994), pp. 802–805.

[15] S. Chen, D. Laefer, and E. Mangina. *State of Technology Review of Civilian UAVs*. In: Recent Patents on Engineering 10 (July 2016), pp. 1–1. DOI: 10.2174/1872212110666160712230039.

[16] The Gene Ontology Consortium et al. *The Gene Ontology knowledgebase in 2023*. In: Genetics 224.1 (Mar. 2023), iyad031. ISSN: 1943-2631. DOI: 10.1093/genetics/iyad031.

[17] Guest Contributor. *Geospatial in MongoDB: A Practical Guide*. URL: https://www.slingacademy.com/article/geospatial-in-mongodb-a-practical-guide-with-examples/ (visited on 03/06/2023).

[18] Apple Developer Documentation. *GKOctree*. URL: https://developer.apple.com/documentation/gameplaykit/gkoctree (visited on 03/05/2023).

[19] Apple Developer Documentation. *GKQuadtree*. URL: https://developer.apple.com/documentation/gameplaykit/gkoquadtree (visited on 03/05/2023).

[20] J.B. Duffy. *GAL4 system in Drosophila: A Fly Geneticist's Swiss Army Knife*. In: Genesis 34.1-2 (Oct. 2002), pp. 1–15. DOI: 10.1002/gene.10150.

[21] K. Engel et al. *Real-Time Volume Graphics*. In: (2006), 518 pages. URL: https://www.vrvis.at/publications/PB-VRVis-2006-001.

[22] ETCS-Project. *BBolt*. URL: https://github.com/etcd-io/bbolt (visited on 03/20/2023).

[23] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. The Factory Method pattern is described in Chapter 5. Boston, MA, USA: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[24] F. Ganglberger et al. *A Data Structure for Real-Time Aggregation Queries of Big Brain Networks*. In: Neuroinformatics 18.1 (2020), pp. 131–149. DOI: 10.1007/s12021-019-09428-9.

[25] F. Ganglberger et al. *BrainTrawler: A Visual Analytics Framework for Iterative Exploration of Heterogeneous Big Brain Data*. In: Computers and Graphics 82 (2019), pp. 304–320. ISSN: 0097-8493. DOI: 10.1016/j.cag.2019.05.032.

[26] F. Ganglberger et al. *Structure-based Neuron Retrieval Across Drosophila Brains*. In: Neuroinformatics 12.3 (2014), pp. 423–434. DOI: 10.1007/s12021-014-9219-4.

[27] F.J. Ganglberger et al. *BrainTACO: an explorable multi-scale multi-modal brain transcriptomic and connectivity data resource*. In: Communications Biology 7.1 (June 2024), p. 730. ISSN: 2399-3642. DOI: 10.1038/s42003-024-06355-7.

[28] M. Goshima. *go-json*. URL: https://github.com/goccy/go-json (visited on 02/24/2024).

[29] M. Gotz and W. B. Huttner. *The Cell Biology of Neurogenesis*. In: Nature Reviews Molecular Cell Biology 6.10 (2005), pp. 777–788. DOI: 10.1038/nrm1739.

[30] M. Guo et al. *SINCERA: A Pipeline for Single-Cell RNA-Seq Profiling Analysis*. In: PLOS Computational Biology 11.11 (Nov. 2015), pp. 1–28. DOI: 10.1371/journal.pcbi.1004575.

[31] R.H. Güting. *An Introduction to Spatial Database Systems*. In: The VLDB Journal 3 (1994), pp. 357–399.

[32] A. Guttman. *R Trees: A Dynamic Index Structure for Spatial Searching*. In: vol. 14. Jan. 1984, pp. 47–57. DOI: 10.1145/971697.602266.

[33] H.J. Haverkort. *An Inventory of Three-Dimensional Hilbert Space-Filling Curves*. In: CoRR abs/1109.2323 (2011). arXiv: 1109.2323.

[34] H.J. Haverkort. *How many three-dimensional Hilbert curves are there?* In: (Oct. 2016).

[35] L. Herzberger et al. *Residency Octree: a hybrid approach for scalable web-based multi-volume rendering*. In: "IEEE Transactions on Visualization and Computer Graphics" 30.1 (Jan. 2024), pp. 1380–1390. ISSN: 1941-0506. DOI: 10.1109/TVCG.2023.3327193.

[36] D. Hilbert. *Über die stetige Abbildung einer Line auf ein Flächenstück*. In: Mathematische Annalen 38.3 (Sept. 1891), pp. 459–460. ISSN: 1432-1807. DOI: 10.1007/BF01199431.

[37] J. D. Hunter. *Matplotlib: A 2D graphics environment*. In: Computing in Science & Engineering 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[38] IBM. *Informix*. URL: https://www.ibm.com/products/informix (visited on 03/05/2023).

[39] Allen Institute. *Allen Human Brain Atlas*. URL: https://human.brain-map.org/ (visited on 03/04/2023).

[40] Allen Institute. *Allen Mouse Brain Atlas*. URL: https://mouse.brain-map.org/ (visited on 03/04/2023).

[41] S. Kamata and Y. Bandoh. *An Address Generator of a Pseudo-Hilbert Scan in a Rectangle Region*. In: Proceedings of International Conference on Image Processing. Vol. 1. 1997, 707–710 vol.1. DOI: 10.1109/ICIP.1997.648017.

[42] S. Kamata, R.O. Eason, and Y. Bandou. *A New Algorithm for N-dimensional Hilbert Scanning*. In: IEEE Transactions on Image Processing 8.7 (1999), pp. 964–973. DOI: 10.1109/83.772242.

[43] Y. Kaymaz et al. *HieRFIT: a hierarchical cell type classification tool for projections from complex single-cell atlas datasets*. In: Bioinformatics 37.23 (July 2021), pp. 4431–4436. DOI: 10.1093/bioinformatics/btab499.

[44] S. Klein et al. *Elastix: A Toolbox for Intensity-Based Medical Image Registration*. In: IEEE Transactions on Medical Imaging 29.1 (2010), pp. 196–205.

[45] T. Kluyver et al. *Jupyter Notebooks - a publishing format for reproducible computational workflows*. In: Positioning and Power in Academic Publishing: Players, Agents and Agendas. Ed. by F. Loizides and B. Schmidt. Netherlands: IOS Press, 2016, pp. 87–90.

[46] S.H. Koslow and S. Subramaniam. *Databasing the Brain: From Data to Knowledge (Neuroinformatics)*. John Wiley & Sons, 2005. ISBN: 978-0471309215.

[47] E.S. Lein et al. *Genome-Wide Atlas of Gene Expression in the Adult Mouse Brain*. In: Nature 445.7124 (2007), pp. 168–176.

[48] H. Luan et al. *The Drosophila Split Gal4 System for Neural Circuit Mapping*. In: Frontiers in Neural Circuits 14 (2020), p. 72. ISSN: 1662-5110. DOI: 10.3389/fncir.2020.603397.

[49] C. Markiewicz et al. *OpenNeuro: An open resource for sharing of neuroimaging data*. June 2021. DOI: 10.1101/2021.06.28.450168.

[50] C. Maurer, R. Qi, and V. Raghavan. *A Linear Time Algorithm for Computing Exact Euclidean Distance Transforms of Binary Images in Arbitrary Dimensions*. In: IEEE Transactions on Pattern Analysis and Machine Intelligence 25.2 (2003), pp. 265–270.

[51] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914439.

[52] W. McKinney. *Data Structures for Statistical Computing in Python*. In: Proceedings of the 9th Python in Science Conference. Ed. by Stefan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[53] Aimee McNamara et al. *Validation of the radiobiology toolkit TOPAS-nBio in simple DNA geometries*. In: Physica Medica 33 (Dec. 2016). DOI: 10.1016/j.ejmp.2016.12.010.

[54] Microsoft. *Azure Costmos DB*. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/query/geospatial-intro (visited on 03/05/2023).

[55] M. Milyaev et al. *The Virtual Fly Brain Browser and Query Interface*. In: Bioinformatics 28 3 (2012), pp. 411–5.

[56] V. Mische. *GeoCouch: Geospatial Queries with CouchDB*. URL: https://www.ibm.com/products/informix (visited on 03/05/2023).

[57] A.V. Molofsky et al. *Astrocytes and Disease: A Neurodevelopmental Perspective*. In: Genes & development 26.9 (2012), pp. 891–907.

[58] G.M. Morton. *A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*. Tech. rep. IBM, 1966.

[59] L. Mroz, H. Hauser, and E. Gröller. *Interactive High-Quality Maximum Intensity Projection*. In: Computer Graphics Forum (2000). ISSN: 1467-8659. DOI: 10.1111/1467-8659.00426.

[60] S.E.A. Münzing et al. *Larvalign: Aligning Gene Expression Patterns from the Larval Brain of Drosophila melanogaster*. In: Neuroinformatics 16 (2017), pp. 65–80.

[61] Bruce Naylor. *A Tutorial on Binary Space Partitioning Trees*. In: (Jan. 2005).

[62] Oracle. *Spatial Database*. URL: https://www.oracle.com/database/spatial/ (visited on 03/05/2023).

[63] D. Osumi-Sutherland, M. Longair, and J.D. Armstrong. *Virtual Fly Brain: An Ontology-Linked Schema of the Drosophila Brain*. In: Nature Precedings (2009).

[64] R.C. Paolicelli and C.T. Gross. *Microglia in Development: Linking Brain Wiring to Brain Environment*. In: Neuron Glia Biology 7.1 (2011), pp. 77–83. DOI: 10.1017/S1740925X12000105.

[65] J.B. Pawley. *Handbook of Biological Confocal Microscopy*. Springer Science & Business Media, 2006. ISBN: 978-0-387-45524-2. DOI: 10.1007/978-0-387-45524-2.

[66] G. Peano. *Sur une courbe, qui remplit toute une aire plane*. In: Mathematische Annalen 36 (1890), pp. 157–160.

[67] R. Pike. *Go Concurrency Patterns*. 2012. URL: https://go.dev/talks/2012/concurrency.slide (visited on 05/16/2022).

[68] *PostGIS*. URL: https://postgis.net/ (visited on 03/05/2023).

[69] D. Purves et al. *Neuroscience*. Sinauer Associates, 2012. ISBN: 0-87893-742-0.

[70] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases - With Application to GIS*. Morgan Kaufmann, 2002. ISBN: 1-55860-588-6.

[71] T. Rohlfing and C.R. Maurer. *Nonrigid Image Registration in Shared-Memory Multiprocessor Environments with Application to Brains, Breasts, and Bees*. In: IEEE Transactions on Information Technology in Biomedicine 7 (2003), pp. 16–25.

[72] S. Saalfeld et al. *CATMAID: Collaborative Annotation Toolkit for Massive Amounts of Image Data*. In: Bioinformatics 25.15 (2009), pp. 1984–1986.

[73] H. Sagan. *A Three-Dimensional Hilbert curve*. In: International Journal of Mathematical Education in Science and Technology 24 (1993), pp. 541–545.

[74] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 0123694469.

[75] H. Schildt et al. *Einführung in die Technische Informatik*. Springer Verlag, 2005.

[76] F. Schulze et al. *Dynamic Spatial Indexes for Large Registered Image and Object Collections*. Technical Report. VRVis Research Center, Dec. 2012.

[77] M. Sharifzadeh and C. Shahabi. *VoR-Tree: R-trees with Voronoi Diagrams for Efficient Processing of Spatial Nearest Neighbor Queries*. In: PVLDB 3 (Sept. 2010), pp. 1231–1242. DOI: 10.14778/1920841.1920994.

[78] G.M. Shepherd. *The Synaptic Organization of the Brain*. 5th. Oxford University Press, 2004. ISBN: 019515956X.

[79] V. Solteszova. *Visual Queries in Neuronal Data Exploration*. Diploma Thesis. TU Wien, June 2009.

[80] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134.

[81] The Go Team. *The Go Memory Model*. June 6, 2022. URL: https://go.dev/ref/mem (visited on 06/17/2022).

[82] C. Y. Ting et al. *Focusing Transgene Expression in Drosophila by Coupling Gal4 with a Novel Split-LexA Expression System*. In: Genetics 188.1 (2011), pp. 229–233. DOI: 10.1534/genetics.110.126193.

[83] S. Tweedie et al. *FlyBase: Enhancing Drosophila Gene Ontology Annotations*. In: Nucleic Acids Research 37 (2009), pp. D555–D559.

[84] VRVis. *Big Data for Neurosciences*. URL: https://www.vrvis.at/en/research/research-projects/big-data-for-neurosciences (visited on 03/01/2023).

[85] VRVis. *BrainStar* - A Data Ccience Platform for the Neurosciences*. URL: https://www.vrvis.at/en/products-solutions/products-licenses/brain (visited on 03/01/2023).

[86] VRVis. *Larvalbrain 2.0*. URL: https://www.vrvis.at/en/research/research-projects/larvalbrain-20 (visited on 03/01/2023).

[87] M.L. Waskomm. *Seaborn: Statistical Data Visualization*. In: Journal of Open Source Software 6.60 (2021), p. 3021. DOI: 10.21105/joss.03021.

[88] J. Weissenböck et al. *Dynamic Volume Lines: Visual Comparison of 3D Volumes through Space-filling Curves*. In: "IEEE Transactions on Visualization and Computer Graphics" 25.1 (Jan. 2019), pp. 1040–1049.

[89] T. Wen. *Json Iterator*. URL: https://jsoniter.com/ (visited on 01/16/2023).

[90] M. Wolf. *Computers as Components. Principles of Embedded Computing System Design*. 5th. Elsevier, 2022. ISBN: 978-0-323-85128-2. DOI: 10.1016/C2020-0-02234-6.

[91] C. Xu et al. *Update Migration: An Efficient B+ Tree for Flash Storage*. In: Database Systems for Advanced Applications. Ed. by Hiroyuki Kitagawa et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 276–290. ISBN: 978-3-642-12098-5.

[92] R. Yagi, F. Mayer, and K. Basler. *Refined LexA Transactivators and Their Use in Combination with the Drosophila Gal4 System*. In: Proceedings of the National Academy of Sciences of the United States of America 107.37 (2010), pp. 16166–16171. DOI: 10.1073/pnas.1005957107.

[93] J. Zhang and S. Kamata. *An N-Dimensional Pseudo-Hilbert Scan Algorithm for An Arbitrarily-sized Hypercuboid*. In: IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society. 2007, pp. 2459–2464. DOI: 10.1109/IECON.2007.4460283.

[94] X. Zhang and Z. Du. *The Geographic Information Science and Technology Body of Knowledge*. 4th ed. Spatial Indexing Entry. D. DiBiase, M. Demers, A. Johnson, K. Kemp, A.T. Luck, Dec. 2017. URL: https://gistbok.ucgis.org/bok-topics/spatial-indexing.