Master's thesis

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

Frederic Mjølsnes

Near Real-Time Indexing and Querying of Spatiotemporal Data in Distributed Key-Value Stores

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2020

NTNU
Norwegian University of Science and Technology

Frederic Mjølsnes

Near Real-Time Indexing and Querying of Spatiotemporal Data in Distributed Key-Value Stores

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2020

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

NTNU
Norwegian University of
Science and Technology

Abstract

The Internet of Things (IoT) is continuously evolving and expanding, and the complexity and volume of the data generated by IoT devices is increasing as a result. IoT devices such as various sensors, cars, smartphones and aerial drones are all equipped to generate large amounts of spatiotemporal data specifying their geographical location at a given point in time. The mass collection and utilization of spatiotemporal data enable a whole set of new services to be deployed, as well as requiring continual improvements in the methods for storing, indexing, processing and querying the data as services demand ever more strict service level agreements. NoSQL systems that are horizontally scalable have been introduced in order to cope with the large volume of data, however, as these are typically one-dimensional data stores, indexing multidimensional data is not a trivial task. Space-filling curves enable the linearization of spatiotemporal data and may allow for the usage of NoSQL systems to support the indexing and querying of spatiotemporal data. It is not clear how well a NoSQL database would support near real-time indexing and querying of spatiotemporal data using space-filling curves and the thesis aims to examine existing methods as well as the characteristics of NoSQL stores in order to answer this.

An examination of methods used in two prevalent systems, GeoMesa and GeoWave, has been performed in order to study how space-filling curves may be used in conjunction with NoSQL stores to index and query spatiotemporal data. Furthermore, to examine how well distributed key-value stores support near real-time indexing and querying of spatiotemporal data a set of requirements has been proposed to provide a framework for evaluation. ScyllaDB, a NoSQL store, has been evaluated with regards to the established requirements, and finally an experimental setup for evaluating the performance of different space-filling curve and key-value store combinations has been proposed.

We found that both GeoMesa and GeoWave provide complete extensible facilities for indexing and querying spatiotemporal data in NoSQL stores. Furthermore, regarding the viability of near real-time scenarios with ScyllaDB it was found that it likely does not meet all the requirements, specifically with regards to latency and range queries. Further work is needed to improve latency guarantees and range query performance before key-value stores based on the LSM-tree can be considered viable.

Sammendrag

"Internet of Things" (IoT) er under stadig utvikling, og kompleksiteten og volumet på generert data øker i takt med at fler IoT enheter kobles til. Enheter som sensorer, biler, smarttelefoner og droner kan alle generere store mengder spatiotemporell data som spesifiserer enhetens plassering på et gitt tidspunkt. Innsamlingen og utnyttelsen av spatiotemporell data på stor skala muliggjør mange nye tjenester, men krever også kontinuerlige forbedringer i metodene for lagring, indeksering, prosessering og spørringer på denne dataen etter hvert som tjenester etterspør strengere servicenivå-garantier. Horisontalt skalerbare NoSQL systemer har blitt introdusert for å håndtere det store datavolumet, men da disse hovedsaklig er utviklet for endimensjonell data i form av nøkkel-verdi-par er indeksering av multidimensjonell data et ikke triviellt problem. Romfyllende kurver kan brukes for å linearisere spatiotemporell data, og muliggjør bruken av endimensjonale NoSQL databaser for indeksering og utføring av spørringer på spatiotemporell data. Det er ikke klart hvor godt en NoSQL database i samband med romfyllende kurver støtter nær sanntid indeksering og spørringer, og denne oppgaven forsøker å svare på dette ved å studere eksisterende metoder og karakteristikker ved NoSQL databaser.

En studie av to eksisterende systemer, GeoMesa og GeoWave, er blitt gjennomført, og metodene som er brukt i disse systemene er blitt kartlagt for å belyse hvordan indeksering og spørringer på spatiotemporell data i NoSQL databaser gjøres i praksis. For å forsøke å svare på hvor godt slike metoder fungerer i nær sanntid scenarier er det blitt foreslått en rekke krav en NoSQL database må imøtekomme for å kunne sies å støtte indeksering og spørringer i nær sanntid. ScyllaDB, en NoSQL database er blitt evaluert opp mot disse kravene, og et sett med eksperimenter er foreslått som kan gi en bedre indikasjon på hvor godt nær sanntid bruksområdet støttes.

Resultatene fra evalueringen av NoSQL databaser og spesifikt ScyllaDB opp mot kravene som ble stillt viser at ikke alle kravene ble oppfyllt av ScyllaDB. Mer spesifikt ble blant annet ikke kravene til forsinkelsestid møtt, og det viste seg usannsynlig at områdespørringer vil kunne utføres innenfor korte tidsrom. For at et NoSQL basert på LSM-trær skal støtte nær sanntid indekseringer og spørringer bør framtidig arbeid være i retning av å forbedre områdespørringer.

Preface

This master thesis was written during 2019-2020 for the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU) under the supervision of Professor Svein Erik Bratsberg. A basic familiarity with database systems, algorithms and data structures is assumed on the part of the reader.

I would like to thank Svein Erik Bratsberg for his assistance and giving me great flexibility to shape the thesis as I saw fit.

In addition, I would like thank my family for always supporting me, and thanks to all the contributors of code to the open source repositories that have been used throughout this thesis.

Frederic Mjølsnes

Trondheim, June 2020

Contents

Summary i

Sammendrag ii

Preface iii

Contents iv

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Motivation 1

1.2 Research Questions and Method 2

1.3 Structure 3

2 Background 4

2.1 Spatiotemporal Databases and Spatiotemporal Data 4

2.2 Scalability 5

2.3 Distributed Key-Value Stores 6

2.3.1 ACID, BASE, CAP Theorem and PACELC 6

2.3.2 Partitioning of Data 9

2.4 Space-Filling Curves 10

2.4.1 Properties of Space-Filling Curves 10
2.4.2 Z-order curve 12
2.4.3 Hilbert curve 13
2.4.4 Spatiotemporal Queries using Space-Filling Curves 13
2.5 Access Methods for Indexing and Querying Linearized Spatiotemporal Data . . . 15
2.5.1 B-Tree 16
2.5.2 B+ Tree 17
2.5.3 Log-Structured Merge-tree 19

3 Methods for Indexing and Querying Spatio-Temporal Data in Distributed Key-Value Stores 21

3.1 GeoMesa 21
3.1.1 Indexing 21
3.1.2 Queries 23
3.2 GeoWave 24
3.2.1 Indexing 24
3.2.2 Queries 26

4 Viability of Near Real-Time Spatiotemporal Indexing and Querying using Distributed Key-Value Stores 27

4.1 Near Real-Time Indexing and Querying of Spatiotemporal Data 27
4.2 Use-Case Description 28
4.3 Requirements 29
4.3.1 Consistency, Availability and Partition Tolerance 29
4.3.2 Scalability 30
4.3.3 Storage 32
4.3.4 Ingestion 33
4.3.5 Querying 33

4.4 Evaluation of ScyllaDB 34
4.4.1 Consistency, Availability and Partition Tolerance 34
4.4.2 Storage 35
4.4.3 Scalability 35
4.4.4 Ingestion 36
4.4.5 Querying 37
4.5 Experimental Setup to Measure Viability of Trajectory Indexing and Querying System 37
4.5.1 Insertion 38
4.5.2 Querying 39
4.5.3 Further considerations 39
4.6 Discussion 40

5 Conclusion and Further Work **42**

5.1 Conclusion 42
5.2 Further Work 43

References **44**

List of Tables

4.1 Characteristics of DRAM, flash and disk based storage mediums. 33

List of Figures

2.1 Illustration of range, kNN and trajectory queries on spatiotemporal point-data. . 5

2.2 Examples of different ways to order a 2-dimensional space using space-filling curves 10

2.3 Three stages of decomposition of a coordinate space using z-ordering 12

2.4 Three stages of decomposition of a coordinate space using the Hilbert curve ordering 13

2.5 B-Tree of order 3 containing 14 keys. 17

2.6 B+ Tree of order 3 containing 14 keys. 18

2.7 LSM-tree with levels 0 ... i 20

3.1 The structure of an Accumulo key generated by GeoMesa for spatiotemporal point-data [31] 22

3.2 Sample Ingest Architecture for Apache Accumulo data store [32] 23

3.3 Sample Query Architecture for Apache Accumulo data store [33] 24

3.4 GeoWave Architecture [37] 25

3.5 GeoWave Key Structure [38] 25

3.6 GeoWave Indexing Strategies [39] 26

4.1 Experimental Setup for Collecting Data about Insertion and Querying Performance 38

Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) [1] is continuously evolving and expanding, and the complexity and volume of the data generated by IoT devices is increasing as a result. IoT devices such as various sensors, cars, smartphones and aerial drones are all equipped to generate large amounts of spatiotemporal data specifying their geographical location at a given point in time. The mass collection and utilization of spatiotemporal data enable a whole set of new services to be deployed, as well as requiring continual improvements in the methods for storing, indexing, processing and querying the data as services demand ever more strict service level agreements.

A wide range of services rely on the storage, processing and querying of spatiotemporal data; however, these services make different demands of the underlying software and hardware depending on their intended use case. This thesis is written with regards to services that require horizontal scalability of system resources, high throughput insertion of spatiotemporal point data collected in real-time from mobile sources and the ability to perform low-latency spatial and temporal range queries on the data. Examples of such services include monitoring crowd flow in areas hit by natural disasters and real-time route planning with coordination between self-driving cars.

As increasing amounts of data is collected from connected devices the underlying system must scale its available resources to accomodate the elevated resource demand. Systems are typically scaled by adding more resources in the form of processing power, memory, disk storage or network bandwidth. There are two main ways in which a system scales its resources [2]. The first method, known as scaling up or vertical scaling, adds more resources to a single machine. The second method, known as scaling out or horizontal scaling, adds more servers to an existing cluster of servers which handle the data in a distributed manner.

Relational database management systems (RDBMS) were originally designed to run on a single server and emphasize normalization of data and enforcing ACID [3] transaction properties, which

are not well suited for distributed systems. Because of this relational databases traditionally resort to vertical scaling. There is ongoing work being done on scaling relational databases horizontally [4], but this work is outside the scope of this thesis.

NoSQL data stores are generally designed for horizontal scalability [5] and as a result they are commonly used in scenarios where such scaling is needed. Real-time indexing and querying of massive amounts of data also typically requires horizontal scaling, which is why NoSQL stores are focused on in this thesis.

In order to store multidimensional spatiotemporal data in one-dimensional key-value stores a space-filling curve can be employed as a linearization technique. There exists a variety of different space-filling curves, and with each having different properties it's not always clear how they impact performance in a given system.

Because of the need for systems that can index and query spatiotemporal data in near real-time this thesis will examine how space filling curves can be used in conjunction with distributed key value stores to index and query spatiotemporal data, as well as attempt to evaulate the viability of using distributed key value stores to index and query spatiotemporal trajectory data in real-time by evaluating ScyllaDB. ScyllaDB [6] is a known as a highly performant wide-column NoSQL store, and due to its performance claims is the subject of evaluation in this thesis.

1.2 Research Questions and Method

Research Questions

- RQ1: How can space-filling curves be used in conjunction with horizontally scalable key-value stores in order to index and query spatiotemporal data?
- RQ2: Is the usage of space-filling curves in conjunction with a horizontally scalable key-value store a viable solution for indexing and querying real-time spatiotemporal data streams with regards to throughput, latency and memory usage?

Method

To attempt to answer RQ1 a review of current state-of-the-art systems will be conducted, with prevalent methods and concepts described.

To attempt to answer RQ2 a set of requirements are established, and a subsequent review of the characteristics of a selected distributed key-value store is performed in order to evaluate how well the requirements are met.

1.3 Structure

The thesis is structured as follows:

- *Chapter 2 Background*, provides the theoretical background and discusses fundamental concepts
- *Chapter 3 Methods for Indexing and Querying Spatiotemporal with Space-Filling Curves in Distributed Key-Value Stores*, provides an overview and discussion of approaches to indexing and querying used in two currently existing systems, GeoMesa and GeoWave
- *Chapter 4 Viability of Near Real-Time Spatiotemporal Systems with Distributed Key-Value Stores*, establishes a set of requirements and subsequently evaluates the characteristics of distributed key-value stores and ScyllaDB against these requirements. In addition a method in the form of a set of experiments for benchmarking the performance of a near real-time system is proposed.
- *Chapter 5 Conclusion and Further Work*, presents a summary of our findings and suggests directions in which further research could be undertaken.

Chapter 2

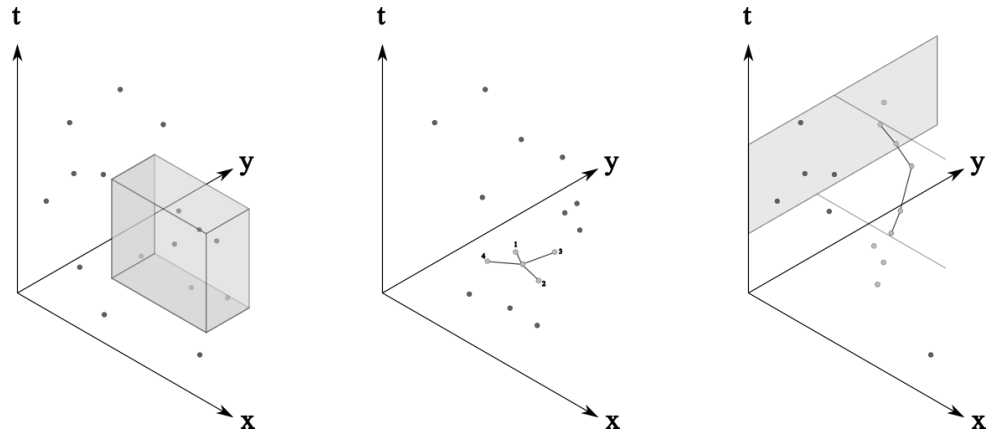
Background

2.1 Spatiotemporal Databases and Spatiotemporal Data

Spatiotemporal databases are specially designed to handle the indexing, storage and querying of spatiotemporal data and objects [7, p. 2150]. The simplest form of spatiotemporal data can be conceived of as a point in a 2- or 3-dimensional space with an associated timestamp, while more complex forms may consist of polygons with associated timestamps. This work deals with spatiotemporal objects of the form $\{x, y, t, p\}$, where x, y, t and p represent *longitude*, *latitude*, *time* and a *payload* such as an UserID respectively. This type of data is referred to in this thesis as spatiotemporal point-data and spatiotemporal trajectory data interchangably.

A spatiotemporal object is typically dynamic in both the spatial and temporal dimensions, meaning that the location of the object can change over time. Because spatiotemporal data is dynamic and can arrive at frequent intervals from several sources, a spatiotemporal database typically needs to support either high update/insertion throughput, depending on the number of sources that are to be indexed.

Spatiotemporal databases should support queries along both the temporal and spatial dimensions and the types of queries that are typically supported are range queries, k-Nearest-Neighbors (kNN) queries and trajectory queries. A range query returns all objects within a given spatial and/or temporal range. A kNN query returns the k nearest neighbours by distance to a given object, usually along the spatial dimension, but can also be done across the temporal dimension. A trajectory query returns all locations within a given temporal range for a given identifiable moving object. An illustration of what the queries look like can be seen in figure 2.1.



(a) Range query on spatial and temporal dimension

(b) kNN query on spatial dimension, k=4

(c) Trajectory query on temporal dimension

Figure 2.1: Illustration of range, kNN and trajectory queries on spatiotemporal point-data.

2.2 Scalability

Scaling a system is the act of adding more resources to a system to meet increased demands, while scalability refers to the system's ability to integrate and use the added resources. Four general types of scalability may be considered [8]. These are *load*, *space*, *space-time*, and *structural scalability*. Taken together these concepts may provide a good indication of how well a given system scales.

- *Load scalability* refers to the degree to which the system can efficiently and smoothly operate under varying degrees of loads. Using various scheduling and load balancing techniques, as well as improving the elasticity of the system enables load scalability.
- *Space scalability* describes the degree to which the system is able to support increased memory requirements, and how the memory requirements increase as the system scales. Using access methods with small space overhead and compression techniques may increase the space scalability of a system.
- *Space-time scalability* refers to how well the system continues to operate as the number of objects in the system increase. Using algorithms and access methods with logarithmic or constant time complexity for operations may enable space-time scalability.
- *Structural scalability* denotes how the structural design choices of a system affect scaling as the number of objects increase. For instance, increasing the bits used for adressing from 32 to 64 bits, as in IPv4 vs. IPv6 increases the structural scalability of the Internet Protocol and systems that depend on it.

2.3 Distributed Key-Value Stores

A distributed data store is a network of connected servers where the data and data access is distributed across the network [9]. Each component in the network may be referred to as a node, and data is typically replicated between nodes to reduce the risk of data loss when nodes fail. A distributed key-value store is a type of distributed data store which stores data in key-value pairs. Distributed key-value stores differ from other distributed databases in that they only provide simple key-value querying mechanisms, rather than full fledged SQL style query mechanisms. This can make supporting advanced queries on distributed key-value stores difficult, but the simple data model allows for robust and easily scalable systems. Because of this lack of full SQL support, distributed data stores of this type are often known as NoSQL databases. An extension of the distributed key-value store is the wide-column store, which can be conceptualized as a two dimensional key-value store. As there are several different implementations of distributed key-value stores, rather than look at specific implementation details we will describe some overall characteristics and concepts used in their design.

2.3.1 ACID, BASE, CAP Theorem and PACELC

ACID (Atomicity, Consistency, Isolation, Durability) is an acronym used to describe a set of desirable properties of transactions in transactional databases. A transaction is a single operation or a set of operations which modify the state of the underlying database system, and ACID is meant to guarantee the validity of the data in the database even if a transaction or operation fails in the face of database errors, network failures and crashes.

- *Atomicity* is a property which guarantees that a transaction either fully succeeds in modifying the state of the system or does not modify it at all. I.e. all operations in a transaction are fully committed and saved, or none are.
- *Consistency* guarantees that any transaction that modifies the state of the database system will bring the system from one valid state to another, and never leave the system in an invalid state.
- *Isolation* guarantees that transactions executed concurrently will leave the database in the same state as if the transactions were executed sequentially.
- *Durability* ensures that any transaction that has been fully committed will remain committed in the face of a system failure, and will be written to the database on system recovery.

Maintaining ACID guarantees in a system typically adds overhead and complexity in the form of synchronization and protocols, even more so when the system is scaled horizontally. Because of this, distributed key-value stores seek to remove the overhead and complexity by modeling themselves on other principles.

BASE (Basically Available, Soft state, Eventual Consistency) is a set of modeling principles that is proposed as an alternative to the ACID transactional model and is the predominant model on which NoSQL systems are based:

- *Basically available* means all nodes in the network are available for read and write operations, but no consistency guarantees are given that a given read will return the latest version of data stored in the cluster, or that a write will persist if a conflicting write has been done somewhere else.
- *Soft state* denotes the fact that without any guarantees of consistency across the system, we can no longer know exactly which state the system as a whole is in at any given point in time, only probabilistically.
- *Eventual consistency* describes the final property, which is that if no new changes in state are made to the system, the system will eventually converge to a globally consistent state.

Proponents of NoSQL systems contend that forgoing the strict ACID transactional requirements, and instead modelling the system on BASE principles allows the distributed database to better scale horizontally, with less performance overhead, lower latency, and higher availability. Modeling systems on the BASE principles however lead to other problems with regards to trade-offs between consistency and availability.

The *CAP (Consistency, Availability, Partition tolerance)* theorem [10] is used to reason about trade-offs in the design of distributed systems such as distributed data stores. The theorem states that a networked system with shared data can only maintain two out of three guarantees at the same time, the guarantees being consistency, availability and partition tolerance:

- *Consistency* is the property of the system to be in a globally consistent state after any insertion or update. I.e. all committed changes in the state of the data will be consistent across all nodes in the network which share said data. This property enables all clients of the system to see the same version of the data at a given point in time.
- *Availability* denotes the degree to which the system is available and responding to requests. With perfect availability the system will remain available for and respond to any request at any given point in time. While availability guarantees that any read request will get a response, depending on the degree of consistency within the system, it does not guarantee that the response contains the latest write.
- *Partition tolerance* denotes the system's tolerance to network partitions, and refers to the degree with which the system can satisfy consistency and availability requirements when the network is partitioned. E.g. a node goes offline or messaging between the nodes is interrupted and as a result a section of the network is partitioned off or becomes unavailable.

In practice no distributed system can rely on all nodes being online and messaging being uninterrupted at all times, and so network partition tolerance is a fundamental requirement for distributed data stores. Furthermore as the number of nodes and interconnects increase with scale, the likelihood of a node failure and messaging failure also increases, making partition tolerance increasingly important as the system scales. In systems where partition tolerance is a requirement the CAP theorem comes into play whenever such a network partition occurs. Consider the case where a node goes offline and a read request is made for data recently written to that node: The system must either refuse or ignore the request, decreasing the availability of the system; alternatively the system can respond with a copy of the requested data stored on a replica node, which increases the availability at the cost of consistency because the replicated data may not be consistent with the data that the offline node was storing.

PACELC (Partition tolerance, Availability, Consistency, Else Latency, Consistency) [11] is an extension of the CAP theorem which states that when network partitioning is not occurring, a tradeoff has to be made between enforcing global consistency and the latency of a system. Enforcing global consistency increases latency because of synchronization overhead as well as increased writes in order to furnish replicas with fresh data. On the other hand, latency can be lowered on the expense of consistency by removing these synchronizations and extra writes.

Further considerations

Database systems can be classified according to which two properties out of consistency, availability and partition tolerance they guarantee. CA focused systems choose consistency and availability over network partition tolerance. These systems do either not support network partitioning at all, or if a network failure occurs the system shuts down completely until it has been manually recovered. CP focused systems choose consistency and partition tolerance over availability. The system will keep its consistency in the face of a network failure, but will become unavailable, requests resulting in either error responses or being ignored. Finally, AP focused systems focus on availability and partition tolerance, and will always try to serve requests in the case of network failures. Even if the response is inconsistent with state stored on other network partitions.

When indexing spatiotemporal data in a distributed key-value store the concepts mentioned above must be taken into account. The system must be partition tolerant in order to scale, so the choice lies between availability and consistency. Combining the CAP theorem and PACELC we find that if the system requires low latency reads and writes and needs to be available at all times, an AP focused system should be chosen. If keeping the system in a consistent state at all times is crucial and availability and low-latency less so, a CP focused system should be chosen. We will consider these properties again when discussing the viability of using distributed key-value stores to index and query spatiotemporal data in real-time.

It is also important to note that in the case that the system is operating normally, both consistency and availability can be satisfied, and it is only during a network partition that the system

must choose between availability and consistency. The choice between consistency, availability and partition tolerance are not binary choices, most systems will support all to a certain degree, the theorem is concerned with the choice between *perfect* consistence, availability and partition tolerance. As Brewer states in *'CAP twelve years later: How the "rules" have changed'* [12]:

> *Although designers still need to choose between consistency and availability when partitions are present, there is an incredible range of flexibility for handling partitions and recovering from them. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application.*

2.3.2 Partitioning of Data

The partitioning of data allow a distributed key-value store to distribute incoming data and requests across different nodes and is typically done by partitioning the data according to the key. Partitioning is employed in distributed key-value stores mainly for load balancing and availability purposes. Partitioning is a complex subject, but the main relevant concepts will be described briefly in this section.

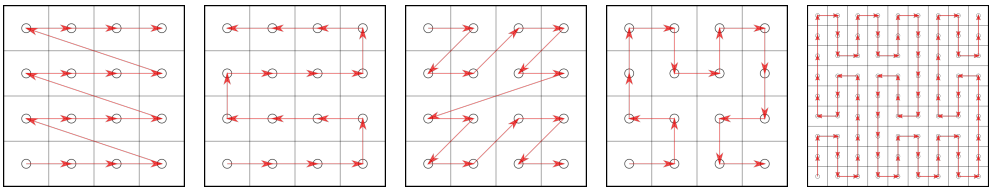
There are several different partitioning schemes that can be employed in distributed systems and distributed key-value stores typically use hash partitioning. In hash partitioning a hash function is applied to incoming keys and the resulting hash is used to determine which node in the network is currently storing, or should store the key-value pair. Each node in the system has a token or a range of tokens associated with it, and a simple linear hash function to determine the correct node in which to store an incoming key would be $token = hash(key) mod N$, where N is the number of nodes in the cluster and *token* is the token associated with a node. This technique works well if the number of nodes remains constant over time, but in a scalable system where nodes go offline and online continually it runs into problems because the same key will be hashed to different nodes when N changes, and a rehashing of all the keys in the database would be necessary to make the hash table consistent again.

To solve the problem of varying node counts consistent hashing with a ring architecture is employed. Consistent hashing can be modeled as hashing values to ranges on a circle, where each node assumes responsibility for a given range on the circle. When a node is added it assumes responsibility for part of it's neighboring nodes ranges, and when a node is removed it distributes its keys to its neighbors. The entire keyspace is divided equally among the available nodes as each node can be responsible for a range of tokens.

The choice of hash function for generating the partitioning tokens is significant. Using a randomizing hash function like Murmur3 will essentially provide system load balancing properties for free, since incoming keys will be distributed evenly across the nodes. However it is a poor choice where range queries must be supported because it destroys the order of the incoming data resulting in a range query having to read from several nodes, then concatenating the result, rather than performing a sequential read. In such a case an order preserving function such

as the identity function may be chosen, which will preserve the global order of keys across the nodes, but will lose the property of load balancing. In practice in systems like Cassandra, a Murmur3 partitioner is used, and ordering of keys is maintained within each node. Murmur3 provides an even distribution of the incoming data across the nodes given that the incoming keys are evenly distributed, since spatiotemporal data is typically skewed, a custom partitioning scheme could be necessary in order to enable good load balancing between the nodes.

2.4 Space-Filling Curves



(a) Row-Major Order (b) Scan Order (c) Z-Order (d) Hilbert Order (e) Peano Order

Figure 2.2: Examples of different ways to order a 2-dimensional space using space-filling curves

Space-filling curves are employed as a linearization technique to map multidimensional data such as spatiotemporal data to one-dimensional values and allows for the indexing and querying of spatiotemporal data using access methods designed for one-dimensional data. For the purposes of this thesis, a space-filling curve is defined as a surjective function which maps each value in a discretized n-dimensional space onto a one-dimensional line while preserving the locality of the input data. The strict mathematical definition of a space-filling curve differs somewhat from the definition commonly employed when linearizing multidimensional data. As a linearization technique the space-filling curve can be conceptualized as being any function that provides an ordered traversal of the points belonging to an n-dimensional space. Some examples of space-filling curves and their traversal order of 2-dimensional squares are shown in figure 2.2. There exists many different space-filling curves and they have different properties. This section discusses locality measures of space-filling curves especially relevant for indexing and querying spatiotemporal data: the z-order and Hilbert curve. And describes how encoding, decoding and querying spatiotemporal data with space-filling curves work.

2.4.1 Properties of Space-Filling Curves

The degree to which a space-filling curve preserves the original locality of the encoded data is an important measure when indexing and querying spatiotemporal data. Preservation of locality refers to how close to each other points in a spatiotemporal space end up when mapped to a one-dimensional space. Haverkort [13] gives a summary of different quality measures used for evaluating how well space-filling curves preserve locality in two dimensions. The ones mainly relevant for use with spatiotemporal data are:

1. Bounds on the worst case or average distance between two points in the plane as a function of their distance along the space-filling curve.
2. Bounds on the average distance between two points along the curve as a function of their distance in the plane.
3. Bounds on the worst case or average number of contiguous sections of the curve needed to cover an axis-parallel query window.

Measure 1 and 2 indicate how well a given curve decodes and encodes spatiotemporal data respectively. Ideally, objects close on the space-filling curve will be close in the spatiotemporal coordinate space and objects close in the spatiotemporal coordinate space will be indexed closely together on the space-filling curve. These measures are particularly important when spatiotemporal data is sorted according to its encoded value because decoded values will then also be located close together in the spatiotemporal space, which is an important property to be able to support kNN-queries by neighborhood propagation well. Measure 3 is be important because it indicates how well a given curve will support range queries. A curve with a high worst case or average number of contiguous sections needed will result in a large number of individual ranges to be scanned, which introduces overhead to the querying process in the form of preparing and executing many separate range queries instead of a few. In addition, with fewer, larger range queries we may benefit from sequential access, while a big number of small range queries will give us an access pattern closer to random access. In practice choosing the best curve is a matter of implementation complexity versus locality preservation. Curves with good locality tend to be more complex to implement and compute, while curves such as the Z-order, with worse locality measures are simpler to implement and compute. Experiments also demonstrate that the gains from implementing a certain curve might be outweighed by the additional computational costs needed in the encoding and decoding process [14], so it is not certain that space-filling curve measures directly translate into real-world performance gains.

Space-filling curves may be classified into two kinds of traversals: discontinuous and continuous [15]. A continous curve guarantees that for any point along the curve, p_i, then the points p_{i-1} and p_{i+1} will at most be located one distance unit away from eachother in the multidimensional space. On the other hand, a discontinuous curve makes no such guarantees, and consecutive points on the curve may be located any distance from eachother in multidimensional space. Typically a continuous curve will generate fewer and larger consecutive ranges than a discontinuous curve, but as mentioned earlier they are typically more complex to implement and require more processing power because operations such as flipping and rotating of the bits are needed in the encoding and decoding process.

Furthermore, space-filling curves can be divided into self-similar and non-self-similar curves. Self-similar curves can be recursively decomposed by subdividing the n-dimensional space into equally sized partitions. Recursive decomposition enables easy parallelization of the decomposition task as well as enabling fast range generation for queries without sacrificing precision. Non-self-similar curves such as the scan-order curve shown in 2.2b can not be decomposed in the same way and as such do not easily derive the benefits of recursive decomposition.

2.4.2 Z-order curve

The Z-order curve shown in figure 2.2c is a discontinuous, self-similar space-filling curve which is commonly used because it is comparatively simple to encode and decode. Because the ordering contains no rotations or flips of cells and is self-similar, encoding values may be achieved by interleaving bits of the spatiotemporal coordinate values, essentially resulting in an equivalent ordering to a depth-first traversal of a quadtree when encoding 2-dimensions, and an equivalent ordering to an oct-tree when encoding 3-dimensions. An illustration of three decompositions of a 2-dimensional space to encode a coordinate pair (x, y) is shown in figure 2.3.

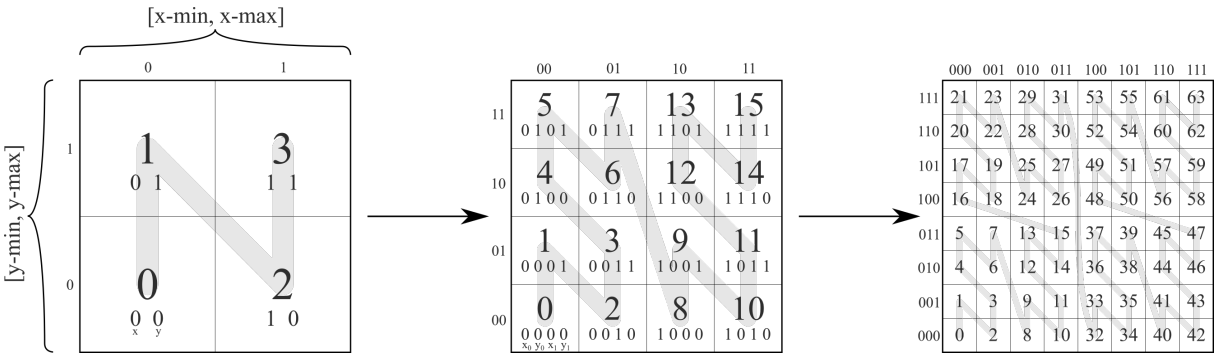


Figure 2.3: Three stages of decomposition of a coordinate space using z-ordering

As we can see from the figure, the curve is recursively decomposed into similar pieces, and each index in the space-filling curve is a result of interleaving the bits the x and y coordinate at a given resolution, starting from the most significant bits of each dimension. This has the desirable property that we can from any given z-order encoded index know which quadrant it is located in just by examining the bits from left to right. This property enables quick encoding of coordinates as well as fast decomposition of range queries.

Expanding the Z-order curve to encode coordinates located in n-dimensional coordinate spaces, like 3-dimensional spatiotemporal coordinate spaces is a simply a matter of interleaving another bit into the sequence for each dimension, and works in the same way. I.e. if we have coordinate (x, y, t), and m bits per dimension; starting from the most significant bit of each coordinate, the encoded value would be $x_0y_0t_0x_1y_1t_1...x_{m-1}y_{m-1}t_{m-1}$.

Decoding an encoded value is done by reversing the process mentioned above, and requires the number of bits used per dimension as well as the number of dimensions in the encoding process to be known. Given an encoded value of a three dimensional coordinate with 4 bits of precision per dimension, 010 100 101 010, corresponds to the coordinate tuple $(x, y, t), x = 0110$, $y = 1001$, $t = 0010 \rightarrow (6, 5, 2)$.

2.4.3 Hilbert curve

The Hilbert curve shown in figure 2.2d is a continuous, self-similar space-filling curve employed for its superior locality preserving properties. Because it is self-similar it can be decomposed using a recursive approach similar to the one described for the Z-order curve, however since it is a continuous curve and the ordering of the points in a given sub-region varies, the encoding and decoding process requires rotation and flip operations in order to calculate the correct index. Figure 2.4 shows three stages of decomposition of a Hilbert curve, and shows how the ordering from smallest to largest index varies within a given sub-region changes depending on the degree of decomposition.

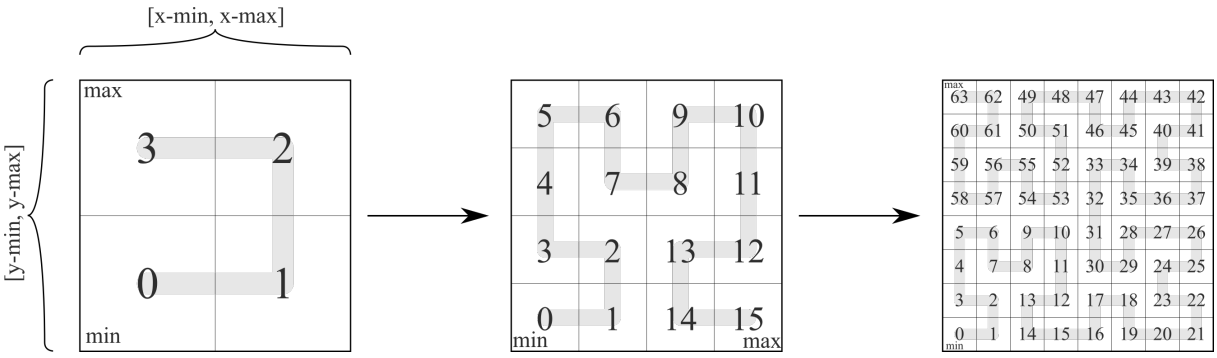


Figure 2.4: Three stages of decomposition of a coordinate space using the Hilbert curve ordering

A Hilbert curve can also be defined for three-dimensions and up. Haverkort explores a subset of the three dimensional variants in [16]. A generalisation of the Hilbert curve for n-dimensional spaces with uneven side lengths, the compact Hilbert Curve has also been developed [17].

2.4.4 Spatiotemporal Queries using Space-Filling Curves

Performing spatiotemporal queries on space-filling curves can be modeled as a four step process, similar to the query processing phases in SQL:

1. *Query formulation*: A spatiotemporal query is formulated in a given query language, then sent to the spatiotemporal data store for a response.
2. *Query parsing, conversion and planning*: The received query is parsed and converted from a spatiotemporal query into a series of one-dimensional queries on the underlying space-filling curves using a query planner. An optional optimization phase on the resulting ranges could also be performed before constructing the final query plan.
3. *Query execution*: The one-dimensional queries are executed against the space-filling curve encoded data in the underlying data store and results are retrieved.
4. *Returning results*: The results are consolidated, optionally converted from one-dimensional to spatiotemporal data using a decoder, and returned to the requester.

The above model suggests some key components that must be present in a system for enabling queries using a space-filling curve:

1. *An encoder, decoder pair*: An encoder turns any multidimensional data into an index along a given space-filling curve. A decoder turns any index along a given space-filling curve into a multidimensional datapoint. Both the encoder and decoder must be specifically implemented for a given space-filling curve, and must have access to the cardinality of the vector space to be encoded, as well as each dimension's length. The encoder and decoder adds a constant cost each time they are used to the querying process, and as such an efficient implementation of both must be considered.
2. *A query planner*: the query planner uses the encoder and decoder, as well as knowledge about the shape of the underlying space-filling curve and the space which it encodes to produce a series of one-dimensional queries that when executed should produce equivalent to the original query being performed in a multidimensional spatiotemporal database. The ideal query planner should compute the fastest and most precise way to query the space-filling curve in the shortest amount of time. In practice these two concerns work against eachother, so a trade-off between accuracy and computation time is usually made.

The above overview of the process enables a description of how specific simple spatiotemporal queries may be performed on data encoded in a space-filling curve:

Range Query

A spatiotemporal query that wants to retrieve all points in a given range can be expressed as a vector describing a bounding box in 3-dimensional space, with the following values:

$$[\, longitude_{min}, latitude_{min}, time_{min}, longitude_{max}, latitude_{max}, time_{max} \,]$$

This vector is then sent to a query planner, which proceeds to generate a query plan consisting of a set of 1-dimensional ranges which cover the region using a given encoder and decoder and a query planning algorithm. Each 1-dimensional range search is then executed against the underlying data store, the results are aggregated and returned.

Polygon Queries

A spatiotemporal query that wants to retrieve all given points within a polygon expressed as a series of points in three dimensions is an extension of the range query. The query planner typically calculates the minimum bounding box of the given polygon, then proceeds to generate the one-dimensional ranges covering the bounding box, and filters out the part of the ranges not covered by the polygon using a point-in-polygon test.

k-NN Queries

A spatiotemporal k-NN query may be performed across either the temporal dimension, spatial dimension, or both. A k-NN query will have different semantics depending on what dimensions is queried. The basic way of performing a k-NN query without any external distance tables is typicall done by a method known as neighborhood propagation. First, the given point that we want to find the nearest neighbors to is calculated to a corresponding index on the space-filling curve. A search rectangle or box is then constructed around this with a given extent. This extent is then decomposed into one-dimensional ranges and the range queries are executed against the underlying data store. If the amount of objects returned from the query is larger than k, the returned objects are sorted according to their distances and the top-k points are returned. If the number of objects is equal to k, then all objects are returned. If the number of objects returned is less than k, the search extent is widened and the process repeats until the k objects are found.

While there exists several variations on how to implement the queries described above, the general concepts of how the queries are performed still apply. The main function of the query planner in all of the above queries is to efficiently deconstruct a given spatiotemporal range into a series of ranges. Specific methods employed for doing this are discussed in the next chapter where we examine how GeoWave and GeoMesa use space-filling curves to enable indexing and querying of spatiotemporal data.

2.5 Access Methods for Indexing and Querying Linearized Spatiotemporal Data

A wide array of access methods have been developed and used for supporting the indexing and querying of spatiotemporal data in spatiotemporal databases[18, 19, 20]. The access methods for spatiotemporal data can be classified in four main categories based on what kind of time frame the data to be indexed exists in. These are spatiotemporal index for the current, recent-past, past and future. They may be further divided into non-parallel and parallel/distributed access methods.

As we are interested in indexing and querying both current and historical data using space-filling curves, the general access methods useful for purpose will be described. In principle any one-dimensional access method can be used in conjunction with space-filling curves as they work by linearizing multidimensional data; however different access methods have various trade-offs with regards to insertion and querying and each of them will be evaluated with regards to using them in conjunction with space-filling curves to index and query large amounts of real-time spatiotemporal data.

2.5.1 B-Tree

B-trees are self-balancing search trees that allow retrieval, insertion and deletion of keys with $O(log_m\, n)$ worst case and average time complexity, where n describes the number of elements being indexed in the tree and m sets the bounds for the number of elements and child nodes for each node [21]. The root of the tree holds $[1, 2m]$ elements, while each node in the tree holds $[m, 2m]$ elements. The root node has $[2, 2m+1]$ children while internal nodes have $[m+1, 2m+1]$ children. The minimum and maximum height h_{min}, h_{max} of a B-tree is given by the formulas $h_{min} = \lceil log_{2m+1}(n+1) \rceil - 1$ and $h_{max} = \lfloor log_{m+1}((n+1)/2) \rfloor$. There are different ways of defining the order of a B-tree, but we will consider the order of a B-tree to be equal to the maximum number of children for internal nodes, $2m + 1$.

Each element in a node consists of a a key which points at a record, k; a value associated with the key, v; and an associated pointer to a child node, $p_{smaller}$. v represents the largest value indexed in the child node pointed to by $p_{smaller}$ as well as the value associated with k. In addition, each node has null element at the end of the array which cointains a pointer, p_{larger}, which points to a child node that indexes keys with values larger than the largest value in the current node.

Retrieving a record associated with a key r in a B-tree is performed by recursively traversing the tree from the root to a leaf node. For each key, k, in a node starting from the smallest we compare r to v. If $r = v_k$ we return the data pointed by k. If $r < v_k$ we follow $p_{smaller}$ to the child node. If we traverse all the elements in the node without finding a key where $r < v_k$ and $p_{larger} \neq nullptr$ we follow p_{larger}. If the first element has $p_{smaller} = nullptr$, and $p_{larger} = nullptr$ we know we are in a leaf node of the tree. If none of the elements in the leaf node has $r = v_k$, the record does not exist.

Inserting a key value pair, kv, in a B-tree starts with finding the leaf node where kv should be inserted. This is accomplished by attempting to retrieve kv from the tree and selecting the leaf node in which the retrieval process ends up. If we encounter a node that has a key where the value of $kv_{node} = kv_{insert}$ we may choose to update the key at the node or simply do nothing and return. If the leaf node marked for insertion contains less than the maxium allowed number of elements for a node we can insert kv directly into the node, keeping the keys in the node sorted by value. If the node is full we need to evenly split the node into two nodes. First we choose the median of the key-value pairs in the node including kv, then the keys with values less than the median are put in the new left node, and the keys with values greater than the median are put in the new right node, with the median key acting as the separation key. The separation key is then inserted into the node's parent, which will be directly inserted if the parent node is not full, or, if the parent node is full a new split will be performed. If the node to be split is the root, a new root node is created above the root increasing the height of the tree. The number of split operations that need to be performed during an insert is bound by the height of the tree, which gives the insertion operation its logarithmic time complexity.

Deleting a key, k, in a B-tree is done by first retrieving k from the tree, deleting it, then rebalancing the tree so that the nodes so that the nodes retain the proper amount of keys and

children. In our case we would typically only insert and retrieve keys from the tree, so the rebalancing process will not be described in further detail.

An illustration of a B-tree where 14 elements were inserted in ascending order is shown in 2.5.

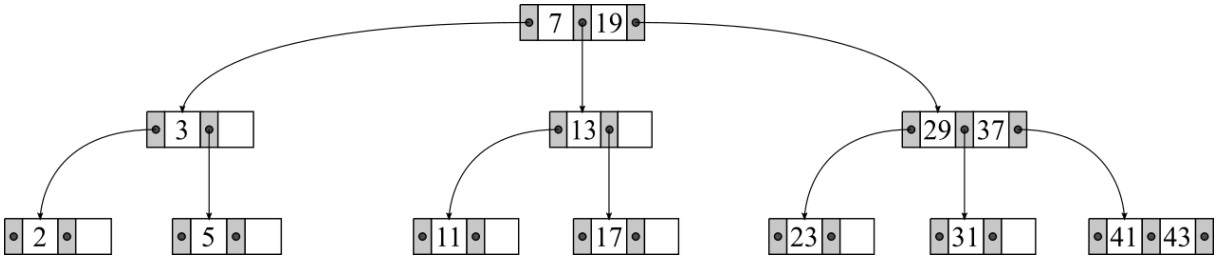


Figure 2.5: B-Tree of order 3 containing 14 keys.

Evaluation

A range query for all keys in a range $[k_0, k_i]$ in a B-tree would have to perform a retrieval operation for each of the keys in the range, which would give a range query a time complexity of $O(i \, log_m \, n)$. This means a range query has linear increase in time complexity as the number of elements in a range increases. Because spatiotemporal range queries on space-filling curves typically involve a high number of range queries, this makes the B-tree ill suited for this purpose.

As mentioned above the insertion complexity of a B-tree increases logarithmically as the number of elements indexed increase. Even though this is a relatively small increase in complexity as the number of elements increased, it will still have an impact on the write throughput as the number of elements increase. For instance, let 10^o be the number of elements indexed by the tree, then the current maximum write throughput, t_c, as a function of the order of magnitude and initial maximum write throughput t_i will be given by $t_c = t_i * (1/o)$. If we reformulate this to express how much initial throughput we need from a system in order to support a throughput at a given order of magnitude we get, $t_i = t_c * o$. It follows that the resources needed to keep a certain throughput level scales linearly with the order of magnitude of keys currently being indexed and the desired insertion throughput.

2.5.2 B+ Tree

The B+ tree [22, p. 652-660] is variant of the B-tree structured similarly in that each node can contain a certain minimum and maximum amount of children and keys. There are some key differences between the two trees which will be described. The first is that a B+ tree stores only keys, and not key-value pairs. The second is that B+ trees store all the keys in the leaf nodes, and root and internal nodes store a copy which indicates the range of the keys stored in the child nodes. The third is that each leaf node contains a pointer to the next leaf node in the series which contain keys with values higher than the node.

As all the keys are stored in the leaf nodes, every retrieval of a key needs to traverse the entire tree to the leaf node, and can not exit early as in the case of the B-tree, this means that a high fanout, i.e. a high number of maximum elements per node, is desirable in order to reduce the overall height of the tree. A higher fanout results in a shorter path to the leaf nodes.

Retrieving a key from a B+ tree is done in a similar fashion to the B-tree. The tree is traversed from the root to the leaf nodes comparing the key, k, to each key in the node, k_n. If $k < k_n$ we traverse the pointer to the child node to the left of the current key. If $k_n < k <= k_{n+1}$ we traverse the pointer to the right child node of the current key. If k is larger than the last k_n in the node we traverse the pointer to the rightmost child node of the current node. This process is repeated until we arrive at a leaf node, at which point the leaf node is returned.

Inserting a key into a B+ tree also works similarly to the B-tree, first finding the leaf node in which the key should be inserted, then inserting the key and rebalancing the tree recursively up to the root if the leaf node is full. There are some implementation differences between inserting in a B+ tree and a B-tree, but for brevity they will be omitted.

The time complexity of inserting and retrieving a key from a B+ tree is $O(log_m\, n)$, the same as in the B-tree. An illustration of a B+ tree of the same number of elements and order as the B-tree in figure 2.5 is shown in figure 2.6.

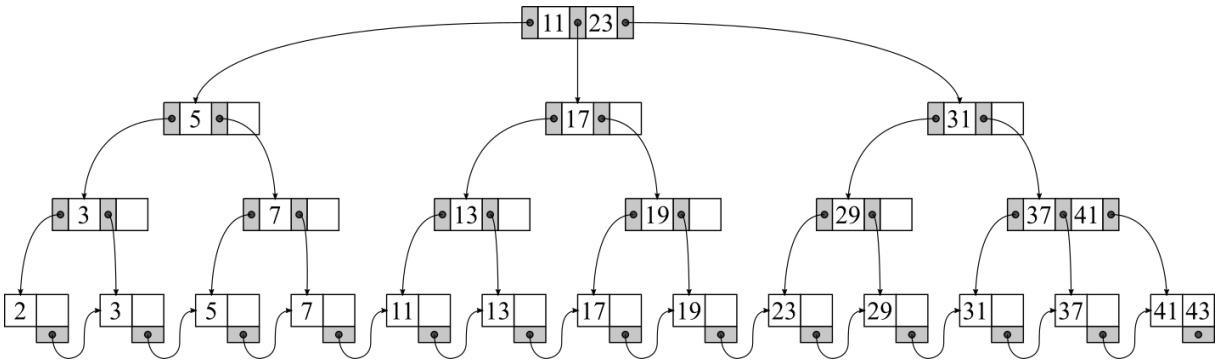


Figure 2.6: B+ Tree of order 3 containing 14 keys.

Evaluation

The B+ tree has one key advantage over the B-tree in relation to range queries. As the leaf nodes are linked together by pointers, only the leaf node containing the lowest key in the range has to be retrieved. The rest of the leaf nodes can be accessed sequentially by following each pointer until the largest key in the range is found. This means that a range query for keys in the range $[k_0, k_i]$ time complexity of $O(log_m\, n\ + i)$ rather than the B-tree's $O(i\, log_m\, n)$, which is a substantial improvement, and makes the B+ tree much better suited for range intensive queries.

The similarity of the B+ and regular B-tree means that the other remarks made in the evaluation of the B-tree also apply to the B+ tree. In addition, the B+-tree may have a higher storage overhead than the B-tree as copies of keys are stored internally.

2.5.3 Log-Structured Merge-tree

The Log-Structured Merge-tree (LSM-tree) is designed to provide an indexing method for situations where a high volume of insertions of key-value pairs are required without sacrificing insertion performance as the number of elements indexed increase [23]. To achieve this, an LSM-tree exploits the characteristics of the underlying storage medium by maintaining separate indexes for each medium. These indexes are intermittently merged using batch strategies when an index is full, or after a certain time period in a process referred to as a compaction.

An LSM-tree maintain its data in at least two levels, where each level represents a tree-like index optimized for the underlying storage medium. In the paper O'Neil et. al. uses an LSM-tree with two levels, c_0 and c_1, for illustration, but there can be any number of levels and more levels are often used in practice. In the case of two levels c_0 resides in main memory, and $c_1 ... c_i$ typically reside on slower, larger storage mediums.

Incoming data is inserted into c_0, which resides in main memory. The tree c_0, has a specified threshold size, and if the record that is inserted should make c_0 reach it's threshold, then the lowest k entries in c_0 are removed from the tree and reorganized into a full leaf node in the tree at c_1. If there are more than two levels, and c_1 is full, the same process is repeated for c_1, compacting the lowest entries in c_1 into a full leaf node for c_2, and so on. As the levels are compacted they grow larger, so $c_0 < c_1 < c_i$. Because the index residing in main memory typically contains comparatively few elements, the insertion time complexity in relation to the total number of elements in the database is $O(1)$, where 1 represents the time it takes to insert a key into the in-memory tree.

The data in secondary storage in an LSM-tree is organized into *runs* of data, sorted by the index key. The runs can be stored as a single file or as a collection of files with non-overlapping key ranges.

Querying an LSM-tree for a key k is done by looking up the key in the in-memory c_0 tree, as well as looking up k in each of the runs, typically using a binary search for each run. The worst case of querying for a key is thus $O(N)$, where N is the number of runs in the LSM-tree. Because the runs reside on slower storage, the base cost to search a run is high. Assuming the tree in c_0 is a B-tree variant with n_{c_0} elements, and the search for each run containing n_r elements is performed using a binary search we thus get a total time complexity of $O(log\, n_{c_0} + N\, log\, n_r$ to look up a single key.

Bloom filters [24] are often used in conjunction with LSM-trees to minimize the associated cost of reading a key. A bloom filter for each run is maintained in-memory, and is consulted before searching for a key in a run. The bloom filter is a compact representation of the elements contained in a given set, and by hashing a key against the filter one can determine whether a given key does *not* exist in the set. This saves having to search for the key in runs where the key does not exist; however, the bloom filter does nothing for determining whether a key *exists* in a set, so these runs will still have to be scanned.

An simplified illustration of the overall structure of an LSM-tree is shown in figure 2.7.

Evaluation

Range queries can be performed in LSM-trees, but suffer from the same limitations in read performance that single key queries do. A naive range query in an LSM-tree is performed by first searching for the lowest key in the range to be searched, then if the key is found in a run, sequentially scanning the run for all the keys until either the range query is exhausted, or a new key search must be performed in order to find the run with the next sequence of keys in the range. If one is searching for large consecutive ranges, and have a relatively low number of large runs to search in, a range query may perform acceptably. However, if the number of ranges is high, and many of the ranges are small ranges, a range query will generally perform worse.

Because of the LSM-tree's good write characteristics distributed key-value stores are often built on this access method. In addition, because the data is stored in different levels asynchronous compression techniques can be employed on the lower levels in order to reduce the size of the database. In the endm choosing a LSM-tree or B+ tree as the underlying access method for indexing and querying spatiotemporal data comes down to a tradeoff between acceptable read and write speeds, or low read speeds and high write speeds. There has been much work done on both improving LSM trees for mixed workloads [25, 26], as well as work showing that highly parallelized implementations of B+ trees can outperform LSM trees on both insertion and querying workloads [27]. An LSM-tree which amortizes insertion cost across several larger less expensive disks and can be highly compressed, will possibly be a good choice of access method if one expects the spatiotemporal workload to be write intensive, and not particularly read intensive.

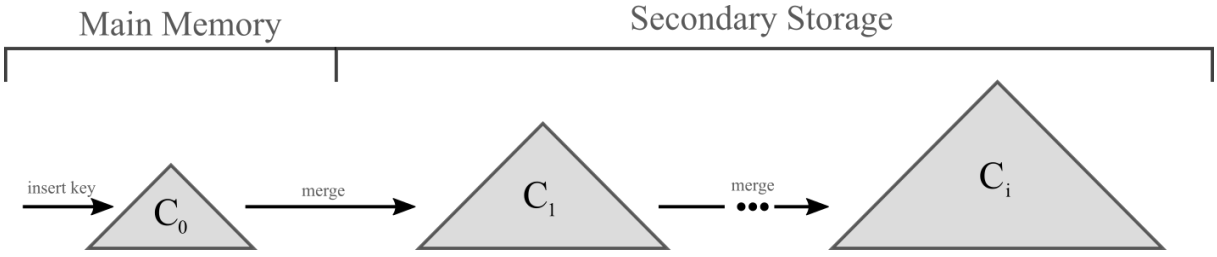


Figure 2.7: LSM-tree with levels 0 ... i

Chapter 3

Methods for Indexing and Querying Spatio-Temporal Data in Distributed Key-Value Stores

This chapter examines how indexing and querying of spatiotemporal data using space-filling curves and distributed key-value stores is accomplished in two existing state-of-the-art systems: GeoMesa and GeoWave. This is done in order to attempt to answer the first research question: *"How can space-filling curves be used in conjunction with horizontally scalable key-value stores in order to index and query spatiotemporal data?"*.

3.1 GeoMesa

GeoMesa [28, 29, 30] is an open source collection of tools licensed under the Apache 2.0 license that, among other things, enable the indexing and querying of spatial and spatiotemporal data with key-value and wide-column stores. GeoMesa does not provide its own underlying data store, but works as a middleware layer that supports several existing open source data stores including Apache Accumulo, Apache Cassandra and Redis. GeoMesa is written in Java/Scala and is maintained by LocationTech, an Eclipse Foundation working group. GeoMesa was originally made to support the efficient indexing and retireval of spatiotemporal point data, but has grown over time to support polygons, secondary attributes and more complex queries. The system uses the Extended Common Query Language as its query language. GeoMesa also includes tools for visualizing spatiotemporal data, but our focus is on how the indexing and querying works.

3.1.1 Indexing

GeoMesa supports spatiotemporal indexing through two kinds of space-filling curves: the Z3 index, which is a three dimensional Z-order curve as mentioned in section 2.4.2; and the XZ3

curve, which is a three dimensional space-filling curve for non-point spatiotemporal data.

The user defines a schema through a SimpleFeatureType object, which defines the attributes of the object that is to be indexed. According to the object attributes specified schema GeoMesa creates indexes to support querying of each attribute. If the schema specifies both a Geometry Point and Date type, GeoMesa will generate a Z3 index based on a three-dimensional Z-curve.

Since time is a potentially unbounded dimension and a space-filling curve must operate in a bounded space, GeoMesa will partition the temporal dimension into periods known as epochs using an approach known as time binning. By default each epoch represents one week, but may also be configured to represent day, month or year intervals. Choosing an epoch length depends on what kind of queries are expected to be executed. If the user expects to execute predominantly short range temporal queries consisting of a few minutes, the index may perform better with an epoch interval of a day. On the other hand if the user expects the workload to consist of mainly long range queries such as several months, the index may perform better with an epoch interval of months or years. This is due to the fact that the larger the epoch, the more keys will have to be scanned within each bin for a temporal range query, and a small epoch interval will allow a query to quickly narrow down to the relevant epoch before scanning.

Tables are split according to the rows, and GeoMesa supports splitting tables according to different attributes. Typically for a Z3 curve the table is split based on the time prefix, i.e. epoch.

Using the schema information, a key structure is constructed that fits the underlying data store. An example of the key-structure used for Accumulo is shown in figure 3.1. As the figure shows the row-key consists of an epoch number, Z3 index and UUID.

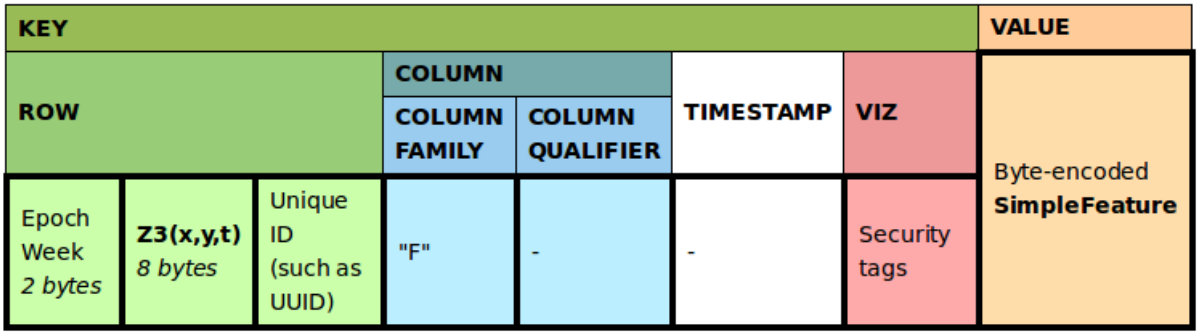


Figure 3.1: The structure of an Accumulo key generated by GeoMesa for spatiotemporal point-data [31]

As GeoMesa is a middleware layer and positions itself in between the data source and underlying data store a variety of ingestion configurations are possible, an example architecture from the GeoMesa documentation is shown in figure 3.2.

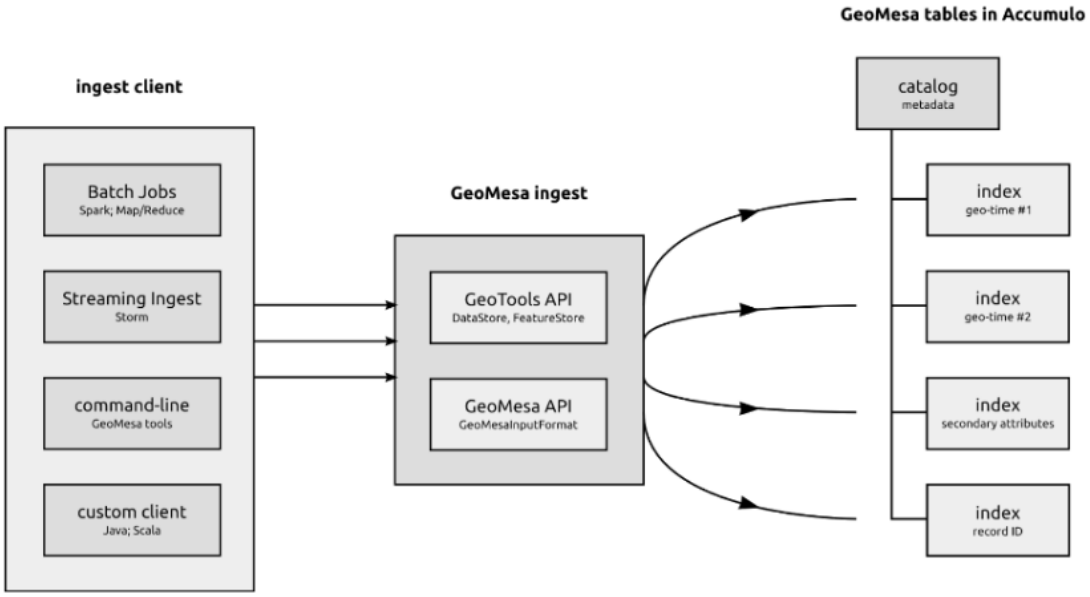


Figure 3.2: Sample Ingest Architecture for Apache Accumulo data store [32]

3.1.2 Queries

GeoMesa supports arbitrary ECQL queries, which include bounding-box/range queries, intersection queries, temporal operators and logical filters. The fact that the user can specify customizable index schemas and that GeoMesa aims to support full ECQL support adds complexity to the query planning process.

Query planning in GeoMesa consists of the following steps. Firstly the CQL query is processed and split based on the indices stored. Then an appropriate index is chosen that can support the query best, i.e. if the query is on spatiotemporal point data the Z3 index will be chosen. After this an abstract query plan for the given index is created by GeoMesa which is translated into a physical query plan compatible with the underlying data store.

GeoMesa selects the appropriate index to use for a given query using one of two strategies:

1. A cost-based strategy, using cached statistics about the datasets in order to select the best index. The stats collected include total count of items in the dataset, histogram and top-k for any indexed attributes.
2. A heuristic strategy, which uses a set of priorities based on the contents of the query in order. E.g. if the query contains spatiotemporal predicates on point data, the Z3 index will be used etc.

A range query on the Z3 curve is performed in the classic manner, by decomposing ranges on the Z3 curve into ranges that overlap with the bounding box specified by the range query. If

the query includes a temporal range, first the range will include only epochs that contain points within the temporal range, and the query is further decomposed into ranges on the Z3 curve in each epoch. K-NN queries are performed by using an expanding query window, which expands the search window until k neighbors have been found.

As with indexing, GeoMesa supports a number of configurations for querying data. An example setup based on the GeoMesa documentation is shown in figure 3.3.

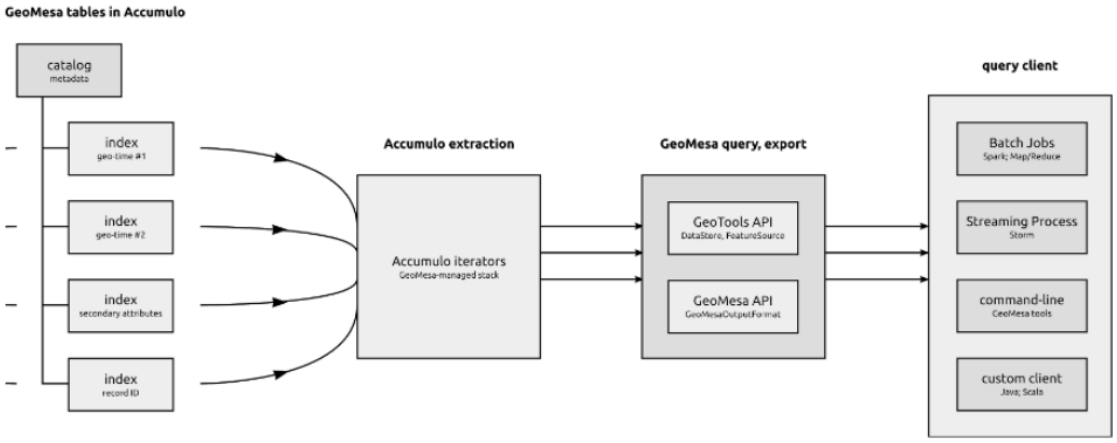


Figure 3.3: Sample Query Architecture for Apache Accumulo data store [33]

3.2 GeoWave

Geowave [34, 35, 36] is an open source set of software that aims to add generalized multidimensional indexing capabilities to key-value stores. It also includes support for indexing and querying spatiotemporal data. Like GeoMesa it is not a key-value store in itself, but acts as a middleware layer to a pluggable key-value store backend. Currently supported data stores are Apache Accumulo, Apache HBase, Apache Cassandra, Amazon DynamoDB, Google BigTable, Redis, RocksDB and Apache Kudu. GeoWave, like GeoMesa, also includes plugins for sharing and visualization of geospatial data, is written in Java/Scala and maintained by LocationTech. Unlike GeoMesa, which requires a separate index for attributes that exceed the spatiotemporal dimensions GeoWave can index data of higher dimensionality by using n-dimensional space-filling curves.

3.2.1 Indexing

GeoWave consists of three main components: data stores, indices and adapters. A data store implements an interface to communicate to an underlying key-value store; the indices are used in order to specify how data should be indexed and retrieved; and the adapters specify how

incoming data should be transformed in order to fit with the schema defined by an index. An overview of the architecture taken from GeoWave's documentation is shown in figure 3.4.

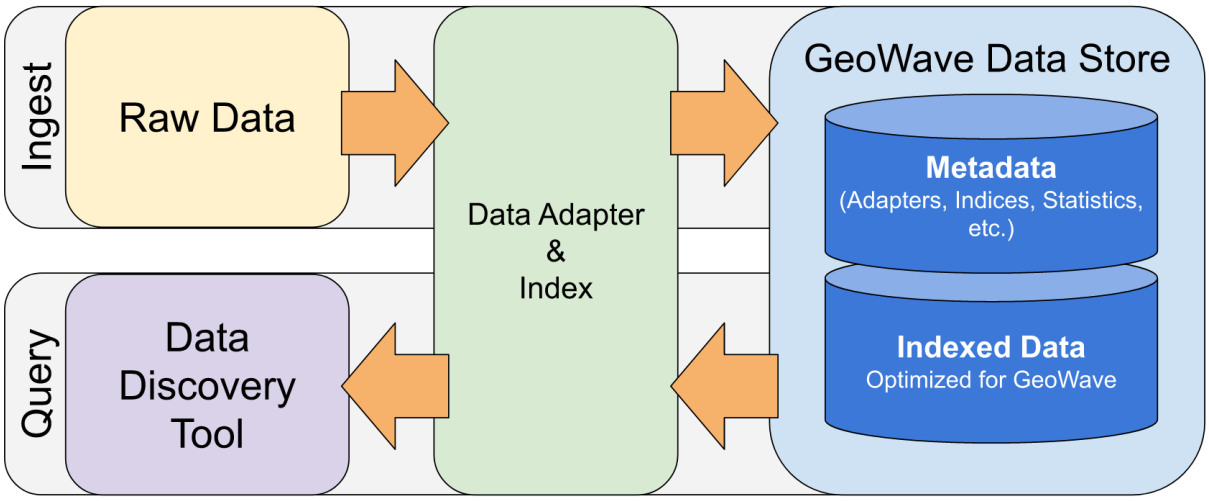


Figure 3.4: GeoWave Architecture [37]

GeoWave uses a default key structure shown in figure 3.5. Of special interest is the partition key and sort key. The partition key is unused by default, but if specified enables GeoWave to partition the spatiotemporal data across different nodes according to a partition strategy which may improve load balance in scenarios where the distribution of the input data is skewed. The currently supported partitioning strategies in GeoWave are round robin and hash-based partitioning, but custom partitioning strategies may be implemented. The sort key is the result of the indexing strategy used to index the spatiotemporal data, and is the value rows will be sorted by.

GeoWave Key						GeoWave Value		
Partition Key	Sort Key (Based on Index Strategy)	Internal Adapter ID	Data ID	Data ID Length	Number of Duplicates	Field Mask	Visibility	Value

Figure 3.5: GeoWave Key Structure [38]

An index in GeoWave consists of a common index model and an index strategy. The index model defines the dimensionality of the data to be indexed. When indexing spatiotemporal data a user first defines three dimensions with given precisions in order to construct the common index model. After the common index model is constructed an indexing strategy is chosen. The indexing strategy determines how the key to be stored in the underlying data store is structured. GeoWave supplies several indexing strategies to choose from, as shown in figure 3.6. Out of these the, TieredSFCIndexStrategy is the primary index strategy GeoWave supports.

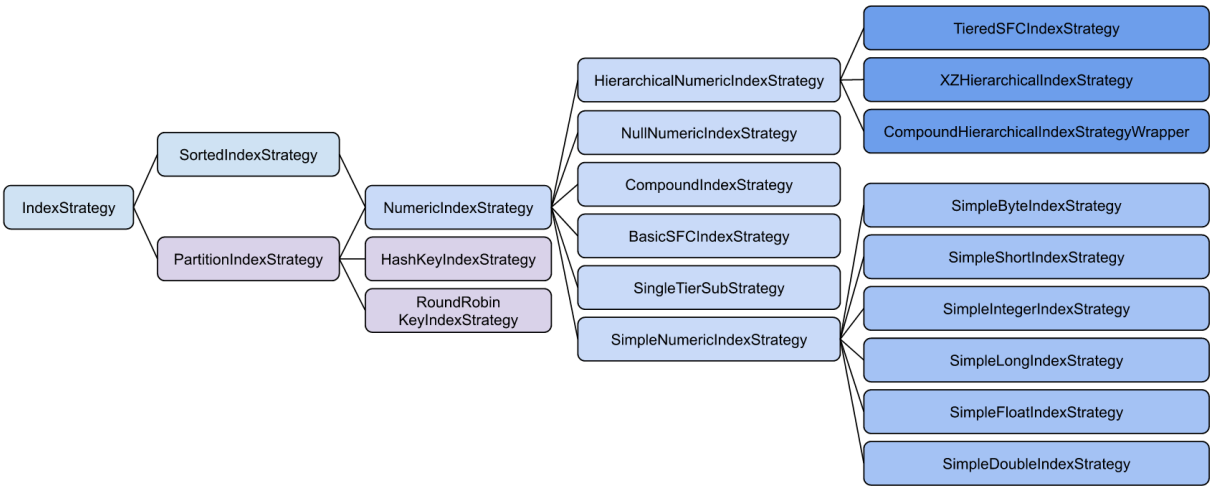


Figure 3.6: GeoWave Indexing Strategies [39]

While GeoWave supports extending the indexing strategies to support custom space-filling curves; two different space-filling curves are implemented by default: the Z-Order curve and the Hilbert curve. The Z-order curve is implemented using the bit interleaving approach mentioned in section 2.4.2 and the Hilbert curve implementation is based on the compact Hilbert curve mentioned in section 2.4.3. The fact that the compact Hilbert curve can have different side lengths for each dimension is exploited in GeoWave in that different precisions can be specified for each value of the data to be indexed. E.g. if the spatiotemporal indexing requires high precision on the spatial dimensions and low precision on the temporal dimension, this is accommodated by GeoWave and can lead to increased query performance since the coordinate space can be made less sparse, leading to fewer and shorter ranges that need to be queried. This allows for the key-length to be only as long as it needs to be to index the data at the wanted resolution for each of the attributes and this directly leads to better space utilization and lower storage requirements when storing a large amount of data. A downside of using the compact Hilbert curve is that it is more complex, and as such requires more computation both when encoding a spatiotemporal point and when decomposing a query into a series of ranges.

3.2.2 Queries

GeoWave supports simple vector queries, but not arbitrary CQL queries like GeoMesa does. A GeoWave vector query is composed of a set of filters and index constraints. The index constraints constrain the dimensional ranges of the queries across the different dimensions of the index; while the filters provide further filtering of the results returned by retrieving the ranges inside the index constraints. E.g. querying GeoWave for all spatiotemporal point data within a given spatial and temporal range would specify three index constraints, one for each spatiotemporal dimension. As with GeoMesa querying, GeoWave first plans the query on the space-filling curve, then translates the query plan into a set of queries that are to be run against the underlying key-value store.

Chapter 4

Viability of Near Real-Time Spatiotemporal Indexing and Querying using Distributed Key-Value Stores

The previous chapter showed how GeoMesa and GeoWave can be used to index and query spatiotemporal data using space-filling curves in conjunction with distributed key-value stores. In principle both can also be used to index streaming spatiotemporal data in near-real time by performing ingestion from a stream-processing software such as Apache Kafka. In this chapter we will attempt to evaluate a key-value store, ScyllaDB, by first defining a set of requirements, and then examining ScyllaDB's properties in relation to these requirements to get an indication of how well it may support near real-time indexing and querying of spatiotemporal data streams. It was originally planned to implement a system and run experiments in order to benchmark its real-world performance, but this had to be dropped because of lack of time and resources. However, we will describe how an experiment evaluating a key-value store's compatibility with real-time streaming data could be performed.

4.1 Near Real-Time Indexing and Querying of Spatiotemporal Data

Real-time systems have constraints on how fast they must deliver responses to requests and update the state of the underlying system. There are three levels of strictness that may be imposed based on the level of guarantee that is needed: *hard*, *firm* and *soft* constraints. If a system operates with hard real-time constraints a broken guarantee will lead to total system failure. If a system operates with firm real-time constraints then a broken guarantee will be tolerated, but the system integrity is degraded and the operation that broke the guarantee must

be discarded. If a system operates with soft real-time constraints and a guarantee is broken then the system integrity is gradually degraded the more the constraint is broken. A near real-time system is typically a real-time system with soft constraints, or larger maximum latency constraints; guarantees may be broken, but the system is designed in order to best meet the required constraints for each operation.

In the context of indexing and querying spatiotemporal data with space-filling curves this means that the indexing operation and querying operations both have constraints on how much time they have to complete, and this may result in high demands with regards to speed on the algorithms used to encode and query the data, as well as the underlying key-value store.

Because the constraints will necessarily differ between use-cases, we will first begin by describing a use-case and then proceed to establish a set of requirements for the given use-case so that we may evaluate how well the requirements are met by the chosen key-value store.

4.2 Use-Case Description

As a primary reason to use a horizontally scalable key-value store is its ease of scalability we will consider a use-case that necessitates scaling of the underlying resources. We consider a common use-case that easily benefits from horizontal scaling: *trajectory indexing*, which essentially consists of continuously indexing spatiotemporal point-data with an associated payload.

A system supporting near real-time trajectory indexing must be able to continuously ingest and index spatiotemporal point-data from an input stream without being overloaded and support low latency queries on the indexed data. Such a system could be useful as a component of any system that consists of continuously moving sensors; such as optimizing traffic flow and collision avoidance for self-driving cars; tracking the movement of large amounts of people in pandemic or natural disaster scenarios; for business intelligence purposes where the business wants to index the movement of customers between websites in cyberspace or even to track the flow of communication packets between nodes in a distributed system.

From the description above we may construct a generalized trajectory indexing use-case with the following characteristics and requirements in order to establish a framework for further evaluation:

1. The system must be able to process a high velocity, high volume incoming data stream consisting of a large number of relatively small spatiotemporal data points within a specified time constraint.

2. The incoming stream of spatiotemporal data should be encoded using a space-filling curve and subsequently inserted into an underlying key-value store. The time spent processing and inserting each tuple should ideally remain constant as the number of tuples indexed increase and must be completed within a specified time constraint.

3. The system should at least support indexing within a bounded temporal dimension. If the temporal bounds of the indexed data are too large to be indexed with satisfying precision, time binning should be supported.

4. The system should at the least support temporal and spatial range queries on the indexed data, and the query response time should remain within a specified time constraint. The query response time should be predictable with regards to the size of the query, and should ideally remain constant even as the number of indexed elements increase.

4.3 Requirements

Based on the characteristics described in the previous section we can derive some requirements for the underlying key-value stores and the system. Specifically we will consider requirements with regards to the CAP theorem, scalability, underlying storage, ingestion and querying.

4.3.1 Consistency, Availability and Partition Tolerance

The system must support network partitioning in order to be scalable, so partition tolerance is a given for the scenario described above. According to the CAP theorem described in section 2.3.1 this means the system designer must choose between an underlying key-store that is either highly available or consistent during a network partitioning.

A BASE system only guarantees eventual consistency, but as trajectory indexing is a scenario where data is continuously written and not intended to be updated this changes how we evaluate the consistency requirement. As data can be assumed be written in an append-only fashion to the system, we may also assume that any replicas of the data that exist in the system will be up to data at any given time after the initial write.

Availability however is a more pressing concern in our scenario. A near real-time system must ensure that data is written to the system and queries responded to within a given time frame, and only highly available systems are able to guarantee this in the case of network partitions.

Assuming we choose to use a highly available and partition tolerant key-value store, the PACELC principle from section 2.3.1 then comes into play. The PACELC principle states that latency and consistency are direct tradeoffs. In our case consistency comes in the form of how many replicas of the data that should be stored so that the most recent writes remain available for querying even when nodes fail. Adding more replicas will increase the amount of writing that needs to be done for each tuple that is indexed and as such might use disk resources that could otherwise be used for indexing new tuples. If the write volume is high and a high number of replicas are made this could possibly decrease the overall throughput of the system. Thus tuning the number of replicas should also be taken into consideration.

4.3.2 Scalability

As mentioned in section 2.2, there exists four main types of scalability: load-, space-, space-time-, and structural scalability. We will consider each of these with regards to the requirements of near real-time trajectory indexing and querying.

Load Scalability

Load scalability refers to how well the system can be scaled to handle increasing loads, and perhaps the most important factor of load scalability is load balancing. Proper load balancing ensures that all nodes in the cluster are evenly stressed, and allows the system to scale to accommodate larger loads without having to resort to overprovisioning of system resources.

When indexing spatiotemporal trajectory data the data distribution and throughput will inherently be skewed. Not all sources of data may be active at the same time which leads to varying levels of throughput that must be supported. In addition the density of the spatiotemporal data is skewed both spatially and temporally with higher densities in certain temporal and spatial ranges. I.e. if tracking the trajectories of cars, a higher density may be observed during rush hour than at other times and cities will have more closely clustered data than rural areas.

Distributing the load evenly across several nodes in a cluster when the data is inherently skewed is not a trivial problem. One should ideally distribute the load such that when ingesting data the write load is evenly distributed across the nodes and when performing range queries each node should not have an inordinate amount of data to search through. Some potential solutions to the problem of load balancing could be to partition the incoming data to different nodes based on the data point's timestamp; its geographical location; or other distinguishing features. Essentially, high density areas in the spatiotemporal space should involve more nodes, while low density areas should involve fewer nodes. As we can expect the distribution of densities not to vary significantly over time over a larger time period such as a month or year, keeping a histogram of the density distribution could also possibly be used in the partitioning scheme. Ultimately, this means that the underlying key-value store must provide support for custom partitioning and load distribution schemes, and this can be considered an essential requirement for the key-value store with regards to load scalability.

Space Scalability

As space scalability refers to how well the system scales with regards to memory requirements, a primary aspect of space scalability is how storage capacity requirements increase as more elements are added to the system. When indexing trajectory data we can roughly predict how many data-points are to be stored in total based on how many sources are being indexed at any time on average and what timespan the system will index. I.e. if we have on average 10^6 concurrent sources of trajectory data, the system needs to provide storage space for around

$8.64 * 10^10$ tuples after 24 hours of indexing.

The large amount of tuples that are involved in trajectory indexing means there are two primary requirements that should be considered with regards to space scalability: The first is to consider the storage overhead of the index that is being used to store the data. As the number of tuples stored increase, the index overhead should ideally remain the same; preferrably increase logarithmically; and in the worst case increase linearly. It follows that the key-value store should use an index that minimizes the storage overhead. However, using the RUM-conjecture[40], we see that in reality the index supported by the key-value store must necessarily make trade offs between write, read and space optimization. As such, the designer of a trajectory indexing system must choose an index that reflects the available resources. I.e. if the index is to be stored in main-memory, the designer could choose a space-optimized index; as main-memory is expensive, but provides fast accesses which may alleviate the added cost of reads and writes. On the other hand,if the index is based on secondary storage, a write or read optimized index may be chosen instead. Ultimately the key-value store must support an indexing scheme that fits the resources available.

The other consideration with regards to space scalability is how well the key-value store supports compression. Compression can be used either to compress the index itself, alleviating storage overhead, or compress the stored data, which will reduce the overall storage requirements of the system as the number of indexed tuples grow. Different key-value stores and indexes support different compression schemes and it is mentioned as it is something that should be supported to provide good space scalability.

Space-Time Scalability

Space-time scalability refers to how well the system continues to operate as the number of objects in the system increase. In order to support good space-time scalability the system should support an index method where inserting new elements and performing range queries on indexed elements, ideally has a time complexity of $O(1)$ and in the worst case better than $O(n)$. As with we mentioned in the section regarding space-scalability, each access method will necessarily make tradeoffs. For example, the LSM-tree which is write/update optimized scales well with regards to insertion, but is performs worse than a B+-tree for point queries and range queries. Because of the necessary tradeoffs, we cannot specify a single access method that the key-value store should support, again, it depends on what resources are available. However, since insertion and range queries are essential components of trajectory indexing, the key-value store should support an index which scales well with regards to these.

Structural Scalability

Structural scalability primarily comes down to choice of space-filling curve, encoding precision and encoding and decoding algorithms. The system must use a space-filling that can represent

the spatiotemporal space that is to be indexed, an encoding precision that allows for the wanted precision, and encoding and decoding algorithms that provide constant, predictable performance.

Choosing a three dimensional Z-curve may be a good choice with regards to constant predictable encoding and decoding performance as the algorithm complexity for encoding and decoding are constant related to the precision chosen for the index. However, a three dimensional Z-curve requires the dimensional side lengths to be equal, which may prove to be suboptimal in scenarios where equal side lengths result in very sparse indexes. As an example, let's say we are indexing spatiotemporal data with a temporal range of 24 hours, and spatial range of the entire surface of the earth. If we were to sample the spatial dimensions at around one meter precision per point at the equator, and the temporal dimension at a precision of one second per point, this would necessitate at least 26 bits for longitude, 25 bits for latitude and only 17 bits for time. In this case a Z-curve would have to use 26 bits for each dimension for a total of 78 bits to represent the data, while an optimal representation would consist of $26 + 25 + 17 = 68\,bits$, which is equivalent to a direct $\approx 14\%$ increase in key size. If we were to use a compact Hilbert curve which supports uneven side lengths, we could reduce this overhead for the cost of added decoding and encoding costs. Ultimately, it comes down to a trade-off between storage capacity and processing power.

The specific requirements with regards to choice of space-filling curve, precision and algorithms will vary based on what spatiotemporal range and precision is needed by the specific scenario. Regardless, the system should support the tuning of these components, i.e. choice of space-filling curve and choice of precision, as well as optimized encoding and decoding algorithms.

4.3.3 Storage

The storage requirements can be divided into the following categories:

1. The system requires enough storage capacity in order to store the index and data.
2. The underlying storage medium must support enough writes and reads per second (IOPS) in order to meet the throughput demands of the system. In addition random read and write throughput should be primarily considered, as in near real-time scenarios we may not necessarily be able to prepare big batches of data for sequential writes.
3. The ratio between throughput and storage capacity should not be unduly skewed. If the storage medium has a small capacity and high throughput, it will be filled quickly and effectively render the IOPS zero as long as it remains full. I.e. if the ratio is too low, we are not efficiently filling the storage space, and if the ratio is too high we are not efficiently utilizing the throughput capabilities of the storage medium over time.

These requirements must be met by the storage medium that the system runs on, table 4.1 describes the overall characteristics of the typical storage choices for reference. As these characteristics are continually evolving the table is only meant as a rough indication of the characteristics, and does not represent precise values.

Storage Medium	Storage Capacity Max	Cost / Storage Capacity	IOPS (4K random R/W)	IOPS/Capacity	Non-Volatile
Random Access Memory (DRAM)	Low (<256GB)	High ≈ 8 $/GB	Very High (<2 million)	7812.5	No
Flash Based Memory (SSD/NVMe)	Medium (<2TB)	Low ≈ 0.1 $/GB	High (<500k)	250	Yes
Disk Based Memory (HDD)	High (<15TB)	Low ≈ 0.04 $/GB	Very low (<150)	0.01	Yes

Table 4.1: Characteristics of DRAM, flash and disk based storage mediums.

As we read from the table for near real-time trajectory indexing, the choice will be between flash based memory and DRAM, as HDD based memory simply is too slow to support the amount of random reads and writes that are needed. While DRAM is expensive, given that trajectory data tuples are typically small, one could possibly use an in-memory solution to index the data. However, because DRAM is volatile, some sort of persistence mechanism must be implemented in addition to prevent loss of data when a node fails. Ultimately, the system should support either a mix of DRAM and flash based storage solutions, or optimized for flash based storage solutions, while support for HDD storage is not a necessity.

4.3.4 Ingestion

There are two primary requirements the system must be able to meet with regards to ingestion of the data. The system must be able to support an insertion rate in proportion to the number of trajectories/sensors that are to be indexed. I.e. indexing of data from n sensors every x seconds means that the system must at least support an insertion rate of $\frac{n}{x}$ $^{tuples}/_{second}$.

In addition, the ingestion latency, i.e. the time for a tuple to arrive until it is written and available for querying, must be lower than the requirement set by the real-time constraints. Furthermore, depending on the strictness of the real-time constraints the system must be able to provide a latency guarantee on the insertion of the data.

4.3.5 Querying

Primarily the system should be able to deliver responses to queries with low latency guarantees, meeting the real-time constraints. Furthermore, the response latency should be predictably correlated to the result size returned by the query, optionally providing automatic scaling of the result set in order to meet the latency guarantees.

Because the trajectory data will be indexed using space-filling curves, meeting latency guarantees is dependent on two main factors. The first is the time it takes to decompose a query into a set of ranges for look-up in the key-value store. The second is how fast and how well the underlying key-value performs range querying. The query planning algorithm is a requirement of the ingestion engine of the system, while range query performance is directly dependent on the characteristics of the underlying key-value store's supported access methods.

The query types that should be supported depend on the specific scenario in which trajectory indexing is performed. At the minimum, point and range queries should be supported, as they are needed in order to support more complex queries such as box-queries, kNN-queries and trajectory queries.

4.4 Evaluation of ScyllaDB

Having established some fundamental requirements for systems to support near real-time indexing and querying of large amounts of trajectory data using space-filling curves, we will now proceed to evaluate a reference key-value store in relation to these requirements in order to get an estimation of how viable it is for near real-time trajectory indexing and querying.

ScyllaDB [6] is a C++ reimplementation of Apache Cassandra with focus on increasing the performance while still keeping Cassandra's data model. It was chosen for this evaluation because both ScyllaDB and Cassandra are well known, widely used and are highly performant. ScyllaDB is optimized for flash storage, and scales vertically as well as horizontally. While ScyllaDB is technically a wide-column store, it is evaluated in this context as a simple key-value store.

The original plan was to implement a real-time trajectory indexing prototype on top of ScyllaDB and run the tests described in section 4.5. Unfortunately, because time and resources did not allow for this, we will use the information available in the documentation as well as third-party benchmarks in order to evaluate the key-value store.

4.4.1 Consistency, Availability and Partition Tolerance

ScyllaDB is an AP type system, prioritising availability over consistency during network partitions. It offers a tunable replication factor, i.e. how many nodes should have a copy of each write, as well as a configurable consistency level. The consistency level determines how many nodes must acknowledge read or write operations before it is considered valid. For example, a consistency level of **ANY** guarantees that the value will be written to at least one node in the cluster before being acknowledged, giving the highest availability, but lowest consistency. On the other hand a consistency level of **ALL** means that a value must be written or read from all replication nodes in the cluster before being acknowledged as valid. There exists a number of consistency level options in between: **QUORUM**, in which a request a valid when a majority of the replicas respond; **ONE**, which is valid when one replica responds; **LOCAL_ONE**, at least one replica in a local data center responds; and several more variations.

The fact that ScyllaDB is highly available and provides tunable replication factor and consistency levels means ScyllaDB meets the requirements set out in the previous section with regards to CAP. In addition the consistency level allows us to fine tune the level we need in order to minimize latency at the cost of consistency. I.e. since we know that values will rarely be updated, a consistency level of **ANY**, or **QUORUM** may provide a lower latency on writes

and reads.

4.4.2 Storage

ScyllaDB is optimized for flash storage and uses RAM for caching and storing node local bloom filters and metadata. The fact that it is optimized for flash storage allows the system to benefit from flash storage characteristics including high IOPS and bandwidth. Furthermore, the size of the cluster can theoretically scale to any size and real world examples of large clusters include Baidu, which runs a petabyte sized cluster [41], so meeting storage capacity requirements is simply a matter of how much hardware one has available.

With regards to throughput, benchmarks performed by IBM [42] show a read/write throughput of $\approx 7.25 \times 10^5$ / 10^6 tuples/second respectively. Furthermore, numbers from Scylla themselves[43], show a cluster of 83 nodes with 28 CPU cores per node achieving over 10^9 reads per second. When inserting tuples into ScyllaDB on a local machine with an SSD supporting $\approx 20kIOPS$, the insertion rate was also found to be $\approx 20ktuples/second$. While these numbers are not independently verified, and the local test indicates nothing more than that Scylla is capable of maxing out the throughput capacity of a mid-range SSD on a local machine, they give an indication that Scylla can support a high throughput rate, and may efficiently exploit the underlying storage medium to meet the throughput requirements for near real-time trajectory indexing.

4.4.3 Scalability

Load Scalability

With regards to load scalability, ScyllaDB supports both custom partitioning schemes as well as the ability to store metadata such as a histogram in auxilliary tables, and use this information to implement more sophisticated load balancing strategies. Without speculating about what an optimal load balancing strategy looks like, ScyllaDB meets the load scalability requirements in that it provides the necessary options for customizing it to the use-case.

Space Scalability

ScyllaDB's storage is based on the LSM-tree described in section 2.5.3. The LSM-tree has a relatively modest index overhead, only the index referring to the compacted tables is needed, which is typically much smaller than the number of indexed tuples depending on how large each table is. Two more pressing concerns with regards to space scalability is the compaction strategy employed and how many replicas are to be stored. With regards to the replication factor, the base storage capacity needed to store the data is equal to $basestoragecapacity \times replicationfactor$ since each tuple is stored multiple times. Considering the compaction strategy employed, the

default compaction strategy, size-tiered compaction, requires having at least 50% free storage space in order to have enough space to perform the table merging in the worst case. This means that with a typical replication factor of 3 and using the default size-tiered compaction strategy, the storage capacity required is 6 times the base storage cost before considering compression. Scylla introduces an incremental compaction strategy [44] which reduces the needed temporary space for compaction alleviating this problem, however this feature is only available in the enterprise version of ScyllaDB, which is not open source. Since we are basing this evaluation on the open source version of ScyllaDB, we can conclude that the open source version of ScyllaDB does not do well with regards to the storage capacity asepct of space scalability, while the enterprise version might do better.

ScyllaDB supports various compression schemes [45], which may reduce the storage requirements by over half at a negligble performance loss when using standard flash storage. In addition tuples that are not intended to be updated can be marked as *frozen*, which enables further compaction of the SSTables containing the tuples. Thus we can say ScyllaDB meets the compression requirements of space scalability.

Space-Time Scalability

As ScyllaDB uses the LSM-tree, the insertion time complexity is amortized $O(1)$, and remains constant as the number of indexed datapoints grow. This means that Scylla meets the requirements of space-time scalability with regards to writes.

When considering the space-time scalability of ScyllaDB with regards to reads we get a more complicated picture. Because the worst case read performance of an LSM-tree is $O(N)$, where N is the number of tuples stored in a node, the read performance and latencies could possibly become too slow to meet the real-time requirements as the number of indexed tuples grow. Furthermore, Scylla does not support efficient range querying as a result of using the Cassandra data model and LSM-tree access method without carefully constructing a data model and using multiple tables and secondary indexes, which may make it difficult to get good read latencies as the database grows. Overall, this means that we can not say for certain that ScyllaDB meets the requirements for read aspect of space-time scalability; an implementation and further testing would be needed in order to ascertain the actual performance.

4.4.4 Ingestion

With regards to the throughput requirements we showed in section 4.4.2 that ScyllaDB can be scaled to meet any throughput requirement, and given the evidence available it exploits the underlying hardware efficiently in order to do so. In ScyllaDB each CPU core in a given node is treated as its own shard, and as a result it may scale efficiently both horizontally and vertically.

ScyllaDB provides no insertion latency guarantees, and the insertion latency is a result of a combination of the performance underlying hardware, the size of the index, ongoing compactions,

network latency and Scylla configuration. Judging from I/O benchmarks provided by Scylla [46] and insertion tests done on local hardware, the insertion latency is typically low, but may jump if concurrency is too high, i.e. data is being queued because throughput is higher than the disks support.

Overall ScyllaDB meets the throughput requirements, and with regards to latency it can meet the latency requirements most of the time if one can make sure there are always enough resources available. However, it does not provide any consistent latency guarantees, which may prove to be an issue for trajectory indexing applications with strict real-time constraints.

4.4.5 Querying

As with ingestion, ScyllaDB does not provide strict query latency guarantees, but may provide consistently low latencies when querying for single point data. We were not able to find any performance benchmarks that indicate how well Scylla performs when querying ranges. Trajectory querying on trajectories stored in a space-filling curve typically involve a large amount of range queries, so without implementing and testing it is not possible to say exactly how good ScyllaDB's range query latencies will be. In general LSM-trees typically have poor read performance with regards to range queries. Given a partitioning scheme where data is evenly distributed among nodes a range query will in the worst case have to read data from all nodes, and for each node, every level of the node's LSM-tree must be searched. When the start of the queried range is found on a given node, an iterator is created which iterates through the SSTable(s) until the end of the range is reached. The results from all the nodes are then merged and returned to the client, adding additional overhead. Overall this evidence indicates that the range query performance with regards to latency of ScyllaDB is poor. We cannot ascertain whether it meets the requirement of low latency range queries, however it seems unlikely. An implementation and benchmarks would have to be performed in order to say for sure.

4.5 Experimental Setup to Measure Viability of Trajectory Indexing and Querying System

While we were not able to perform these experiments, the experimental setup that was intended to be performed is described in this chapter in order to provide a framework within which a system intended to index and query trajectory data using space-filling curves might be evaluated. Figure 4.1 illustrates the overall experimental setup.

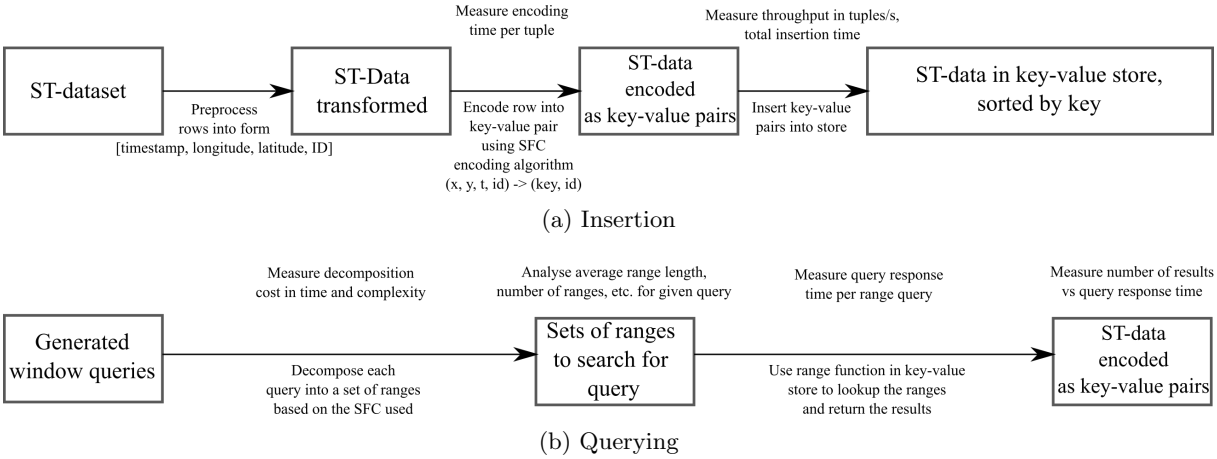


Figure 4.1: Experimental Setup for Collecting Data about Insertion and Querying Performance

The experimental setup consists of two experiment groups: insertion, shown in figure 4.1a, and querying, shown in figure 4.1b. Each of them consist of data being transformed by a set of functions, and a set of measures which measure characteristics of the data as it is being transformed, as well as measuring the performance of each function. Overall, this experimental setup should be able to both evaluate the performance a given space-filling curve and it's associated encoding, decoding and decomposition algorithms, as well as measuring the performance of the underlying key-value store.

4.5.1 Insertion

The insertion experiment takes as input one or more spatiotemporal datasets, the datasets could either be synthesized or be real world datasets such as the T-Drive Taxi Trajectories[47, 48], or GeoLife Trajectories[49, 50, 51]. The benefit of using a real world dataset is that it will inherently be skewed, and as such might give a better indication of real-world performance than a synthetic dataset.

After loading the input data it is pre-processed into a form that is ready for insertion. Each tuple in the transformed dataset is then encoded into a key-value pair using a corresponding space-filling curve encoding algorithm, and the average encoding time per tuple is measured. This gives us an idea of how performant the encoding algorithm is, as well as establishing an upper bound on the tuple insertion, as tuples cannot be inserted faster than they are encoded. After the tuples are encoded they are then inserted into the key-value store and total insertion time, average throughput as well as maximum write latency is measured. This indicates how well the insertion part of the system is working.

4.5.2 Querying

To generate the window queries that are used as input in the querying experiment the encoded ST-dataset from the insertion step is used. The dataset is sorted by key, then by iterating through the dataset n queries with increasing ranges is generated based on how many results should be in the query. I.e. first a set of n range queries each containing r results may be generated by iterating over the entries of the dataset, then setting a start and end range for every r rows. This process can be repeated in order to produce several sets of range queries with different result sizes. After this, each encoded range should be decoded into a x, y, t using the inverse of the space-filling curve function used in the insertion step.

When the generated window queries are properly prepared each of the queries are then decomposed into a set of ranges using a decomposition algorithm for the chosen space-filling curve, each decomposition is measured by the time spent to decompose the query, the spatiotemporal size of the query, and how many ranges were produced. Because of the different locality measures of space-filling curves, a Hilbert Curve will produce fewer larger, contiguous ranges on the cost of more computation time, while a Z-curve will be fast, but produce more ranges to be queried. Using these measures will give an indication of how well the chosen decomposition algorithm and space-filling curve is performing compared to others.

After generating the sets of range queries they are subsequently executed against the key-value store. The average query response time for each range query based on the results returned should be measured, as well as the total response time vs. how many results are returned. The results show how the key-value store performs with range queries with regards to results returned, as well as the performance of individual range queries.

4.5.3 Further considerations

The experimental setup described above is not complete. For a more complete experiment one would have to first set up a constant streaming data source, such as Apache Kafka which continuously feeds tuples into the system in order to measure how performance changes over time. In addition, one should consider the performance implications of multitenancy query scenarios, as well as how performance is impacted by concurrently inserting and running queries. The complexity and amount of variables in the experiment quickly increase, and as such the experimental setup above can be used in order to give a preliminary indication of how a given system will perform and identifying possible bottlenecks before instantiating a complete production test environment.

4.6 Discussion

After qualitatively evaluating Scylla against the requirements set up in section 4.3 it was found that Scylla does not meet all the requirements. The lack of latency guarantees precludes any scenario where such guarantees are needed, the large storage overhead means overprovisioning storage is necessary which adds to the overall cost and maintenance overhead of the system, and the likely poor range query latencies mean near real-time querying is probably unfeasible. While it is possible that these issues may be mitigated with proper tuning, because of the underlying LSM-tree's characteristics they will likely remain a problem, especially considering range query latencies. In order to ascertain the exact performance and viability of Scylla a cluster would have to be set up and experiments performed which we unfortunately were not able to perform. Ultimately, as the evidence stands Scylla does not present itself as a good alternative for near real-time trajectory indexing and querying using space-filling curves. Alternative key-value stores which do not use LSM-trees, and solutions that work directly on distributed file systems like HDFS might prove to be more viable, but were not evaluated.

While researching, two alternative methods of interest were found that according to their authors support near real-time spatiotemporal indexing and querying:

First, a paper by *Park, Ko and Song* [52] suggests a *Parallel Insertion and Indexing Method for Large Amount of Spatiotemporal Data Using Dynamic Multilevel Grid Technique*, and implements a working prototype using the Hadoop Distributed File System and Apache Accumulo. The system manages to provide ingestion performance of skewed spatiotemporal data that is much faster than GeoMesa's ingestion throughput,providing around 8×10^5 tuples per second versus 10^5 tuples per second respectively using a 9 node cluster. However, while scaling better than GeoMesa when performing range queries, the range query throughput experiments is limited to queries with a small amount of results returned, at only 120 on average per query. As there is no data showing how the system performs with larger range queries, this indicates that performing range queries may be a weakness of this system as well.

Ruichi, Cai et. al. propose a system in their paper *DITIR: distributed index for high throughput trajectory insertion and real-time temporal range query* [53]. DITIR uses HDFS as the underlying storage medium, as well as a B+-trees with a templatized insertion scheme in order to reduce the insertion overhead associated with B+-trees. The spatial component of the incoming tuples is encoded using a z-order curve and inserted in a B+-tree, and the B+-trees are subsequently flushed to storage periodically, where each b+-tree is responsible for a given temporal range. A metadata 2-dimensional R-tree provides an indexing scheme which correlates spatial and temporal ranges with the appropriate chunks on the underlying filesystem. Overall the system provides an insertion throughput of around 7.5×10^5 tuples per second on a 10 node cluster according to its authors. This is an acceptable ingestion performance, but not markedly different to ingestion speeds achieved by other systems, such as Scylla. Their query range results show a consistent query latency of around 500 ms per query, which can be regarded as good for certain applications, but the authors do not provide enough details in their paper to assess the range query performance, i.e. they only specify the selectivity of the query and not the total amount

of results returned by the query. In addition the system is optimized for temporal queries, first breaking down the query by temporal range, and subsequently filtering the spatial range. This approach will work well for smaller temporal ranges, i.e. a smaller number of B+-trees are involved in the query, but for a range query with a large temporal range and a small spatial range, such an approach might lead to large response times as a large amount of separate B+-trees must be consulted.

Both examples indicate that the bottleneck for near real-time indexing and querying does not lie with ingestion throughput, but rather with query latencies as with ScyllaDB. This indicates that query latencies is a concern that must be especially considered, and is perhaps more important than pure write throughput.

Chapter 5

Conclusion and Further Work

5.1 Conclusion

Two research questions were posited for research in this thesis:

- RQ1: How can space-filling curves be used in conjunction with horizontally scalable key-value stores in order to index and query spatiotemporal data?
- RQ2: Is the usage of space-filling curves in conjunction with a horizontally scalable key-value store a viable solution for indexing and querying real-time spatiotemporal data streams with regards to throughput, latency and memory usage?

To answer RQ1 two existing systems, GeoMesa and GeoWave, that use space-filling curves to index and query spatiotemporal data on distributed key-value stores were identified. These systems were examined and the methods they use were described. Both systems can be extended with support for additional key-value stores, as well as implementing custom space-filling curves, with GeoWave being the more modern and extensible of the two. GeoMesa supports full CQL queries, while GeoWave supports more basic range queries. By showing the methods and architectures used we showed different ways in which space-filling curves can be employed in conjunction with key-value stores, and that the open source resources exist to implement such a system with relatively low development investment.

In order to answer RQ2 a set of requirements were established that should be met by a system for it to be viable. Furthermore a reference key-value store, ScyllaDB, was evaluated against these requirements. Scylla was found to meet the requirements related to consistency, availability, partition tolerance, load scalability and insertion throughput. However it did not meet the requirements with regards to latency guarantees on reads and writes. In addition space-scalability was poor, requiring a large storage overhead equal to double the size of the indexed data to enable compactions to take place. In addition range query support was found to be

lacking, with range queries typically involving large response times depending on the number of results in the query.

It was found that Scylla provides tuning options that could mitigate these issues, but as we were unable to implement and run tests we can not ascertain whether it will actually alleviate the issues. An experimental setup was alsp proposed to enable evaluation of different design combinations when performing near real-time indexing and querying of spatiotemporal data. This setup can be used to evaluate both the effect of choosing a certain space-filling curve and associated encoding, decoding and decoding algorithms. It can also be used to evaluate how well a given key-value store and different tunings of the store will perform.

Finally, two alternative implementations were examined and found to provide similar insertion performance to Scylla, while the range query latency could not be ascertained fully given the information available in the papers.

Overall, we conclude that if the range query latency issues can be overcome, and latency guarantees can be made, then horizontally scalable key-value stores in conjunction with space-filling curves may be a viable solution. In the case of LSM based key-value stores the write amplification and storage overhead is also a concern with regards to the viability, however storage capacity is relatively cheap, and the latency issues are the bigger barrier.

5.2 Further Work

The most pressing further work involves actually implementing a system and properly measuring the impact choice of space-filling curve, associated algorithms and key-value store has on the performance of the system. Different space-filling curves result in different amounts of ranges that must be queried that could mitigate the bad range query latency. In addition implementing a proper experimental setup on a multi node cluster would enable relatively direct comparison of performance between key-value stores, and it would be interesting to see how B+-tree based key-value stores compare to LSM-tree based key-value stores.

Finding ways to improve range query support, as well as reducing query latencies has emerged as an important avenue for further work. Investigating how more complex queries such as prediction of future trajectories, kNN queries and calculating intersections between trajectories are also directions for further work.

Finally, other types of distributed databases such as SQL based systems like PostgreSQL could be evaluated as possible candidates. It would be especially interesting to see whether timeseries databases can efficiently support trajectory indexing and querying of spatiotemporal data.

References

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey". In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[2] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced computer architecture and parallel processing*. Vol. 42. John Wiley & Sons, 2005, p. 66.

[3] Theo Haerder and Andreas Reuter. "Principles of transaction-oriented database recovery". In: *ACM computing surveys (CSUR)* 15.4 (1983), pp. 287–317.

[4] ABM Moniruzzaman. "Newsql: Towards next-generation scalable rdbms for online transaction processing (oltp) for big data management". In: *arXiv preprint arXiv:1411.7343* (2014).

[5] Rick Cattell. "Scalable SQL and NoSQL data stores". In: *Acm Sigmod Record* 39.4 (2011), pp. 12–27.

[6] ScyllaDB Inc. *ScyllaDB Documentation*. URL: `https://docs.scylladb.com/` (visited on 02/15/2020).

[7] Hui Xiong, Xun Zhou, and Shashi Shekhar. *Encyclopedia of GIS*. 2017.

[8] André B Bondi. "Characteristics of scalability and their impact on performance". In: *Proceedings of the 2nd international workshop on Software and performance*. 2000, pp. 195–203.

[9] Yaniv Pessach. *Distributed storage: concepts, algorithms, and implementations*. 2013.

[10] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[11] Daniel Abadi. "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story". In: *Computer* 45.2 (2012), pp. 37–42.

[12] Eric Brewer. "CAP twelve years later: How the" rules" have changed". In: *Computer* 45.2 (2012), pp. 23–29.

[13] Herman Haverkort and Freek van Walderveen. "Locality and bounding-box quality of two-dimensional space-filling curves". In: *Computational Geometry* 43.2 (2010), pp. 131–147.

[14] Boris Muratshin. *Hilbert curve vs Z-order*. URL: `https://weekly-geekly.github.io/articles/340100/index.html` (visited on 03/11/2020).

[15] Herman Haverkort. "Sixteen space-filling curves and traversals for d-dimensional cubes and simplices". In: *arXiv preprint arXiv:1711.04473* (2017).

[16] HJ Haverkort. "How many three-dimensional Hilbert curves are there?" In: *Journal of Computational Geometry* 8.1 (2017), pp. 206–281.

[17] Chris H Hamilton and Andrew Rau-Chaplin. "Compact Hilbert indices: Space-filling curves for domains with unequal side lengths". In: *Information Processing Letters* 105.5 (2008), pp. 155–163.

[18] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. "Spatio-temporal access methods". In: *IEEE Data Eng. Bull.* 26.2 (2003), pp. 40–49.

[19] Long-Van Nguyen-Dinh, Walid G Aref, and Mohamed Mokbel. "Spatio-temporal access methods: Part 2 (2003-2010)". In: (2010).

[20] Ahmed R Mahmood, Sri Punni, and Walid G Aref. "Spatio-temporal access methods: a survey (2010-2017)". In: *GeoInformatica* 23.1 (2019), pp. 1–36.

[21] Rudolf Bayer and Edward McCreight. "Organization and maintenance of large ordered indexes". In: *Software pioneers*. Springer, 2002, pp. 245–262.

[22] Ramez Elmasri and Sham Navathe. *Fundamentals of Database Systems 6 ed.* 2010.

[23] Patrick O'Neil et al. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[24] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[25] Jin Wang et al. "PLSM: a highly efficient LSM-tree index supporting real-time big data analysis". In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. IEEE. 2013, pp. 240–245.

[26] Dejun Teng et al. "A low-cost disk solution enabling LSM-tree to achieve high performance for mixed read/write workloads". In: *ACM Transactions on Storage (TOS)* 14.2 (2018), pp. 1–26.

[27] Muhammad A Awad et al. "Engineering a high-performance GPU B-Tree". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 2019, pp. 145–157.

[28] James N Hughes et al. "Geomesa: a distributed architecture for spatio-temporal fusion". In: *Geospatial Informatics, Fusion, and Motion Video Analytics V*. Vol. 9473. International Society for Optics and Photonics. 2015, 94730F.

[29] Eclipse Foundation LocationTech working group. *GeoMesa Documentation*. URL: `https://www.geomesa.org/documentation/` (visited on 02/10/2020).

[30] LocationTech. *LocationTech working group, Eclipse Foundation*. URL: `https://github.com/locationtech/geomesa` (visited on 02/10/2020).

[31] Eclipse Foundation LocationTech working group. *GeoMesa Accumulo key structure*. URL: `https://www.geomesa.org/documentation/_images/accumulo-key.png` (visited on 03/09/2020).

[32] Eclipse Foundation LocationTech working group. *Sample Ingest Architecture in GeoMesa*. URL: https://www.geomesa.org/documentation/_images/sampleIngestArch.png (visited on 03/09/2020).

[33] Eclipse Foundation LocationTech working group. *Sample Query Architecture in GeoMesa*. URL: https://www.geomesa.org/documentation/_images/sampleQueryArch.png (visited on 03/09/2020).

[34] Michael A Whitby, Rich Fecher, and Chris Bennight. "Geowave: Utilizing distributed key-value stores for multidimensional data". In: *International Symposium on Spatial and Temporal Databases*. Springer. 2017, pp. 105–122.

[35] Eclipse Foundation LocationTech working group. *GeoWave documentation*. URL: https://locationtech.github.io/geowave/overview.html (visited on 02/10/2020).

[36] Eclipse Foundation LocationTech working group. *GeoWave GitHub Repository*. URL: https://github.com/locationtech/geowave (visited on 02/10/2020).

[37] Eclipse Foundation LocationTech working group. *GeoWave Architecture Overview*. URL: https://locationtech.github.io/geowave/images/architecture_overview_dev.svg (visited on 03/09/2020).

[38] Eclipse Foundation LocationTech working group. *GeoWave Key Structure*. URL: https://locationtech.github.io/geowave/images/keystructure.svg (visited on 03/09/2020).

[39] Eclipse Foundation LocationTech working group. *GeoWave Indexing Strategies*. URL: https://locationtech.github.io/geowave/images/IndexStrategyHierarchy.svg (visited on 03/09/2020).

[40] Manos Athanassoulis et al. "Designing Access Methods: The RUM Conjecture." In: *EDBT*. Vol. 2016. 2016, pp. 461–466.

[41] Zhangmei Li and Jeff; Baidu Security R&D Department Liu. *How Baidu Runs Scylla on a Petabyte Level Big Data Platform*. URL: https://www.slideshare.net/ScyllaDB/scylla-summit-2017-how-baidu-runs-scylla-on-a-petabytelevel-big-data-platform (visited on 03/20/2020).

[42] IBM. *IBM ScyllaDB Performance Benchmark*. URL: https://developer.ibm.com/articles/l-performance-scylla/ (visited on 03/20/2020).

[43] ScyllaDB. *ScyllaDB Performance Benchmark*. URL: https://www.globenewswire.com/news-release/2019/11/05/1941144/0/en/ScyllaDB-Smashes-Performance-Record-Hits-1-000-000-000-RPS.html (visited on 03/20/2020).

[44] Inc. ScyllaDB. *ScyllaDB Incremental Compaction Strategy*. URL: https://www.scylladb.com/2020/01/16/maximizing-disk-utilization-with-incremental-compaction/ (visited on 03/20/2020).

[45] Inc. ScyllaDB. *Compression in ScyllaDB*. URL: https://www.scylladb.com/2019/10/07/compression-in-scylla-part-two/ (visited on 03/20/2020).

[46] Inc. ScyllaDB. *The Scylla I/O Scheduler*. URL: https://www.scylladb.com/2018/04/19/scylla-i-o-scheduler-3/ (visited on 03/20/2020).

[47] Jing Yuan et al. “Driving with knowledge from the physical world”. In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2011, pp. 316–324.

[48] Jing Yuan et al. “T-drive: driving directions based on taxi trajectories”. In: *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*. 2010, pp. 99–108.

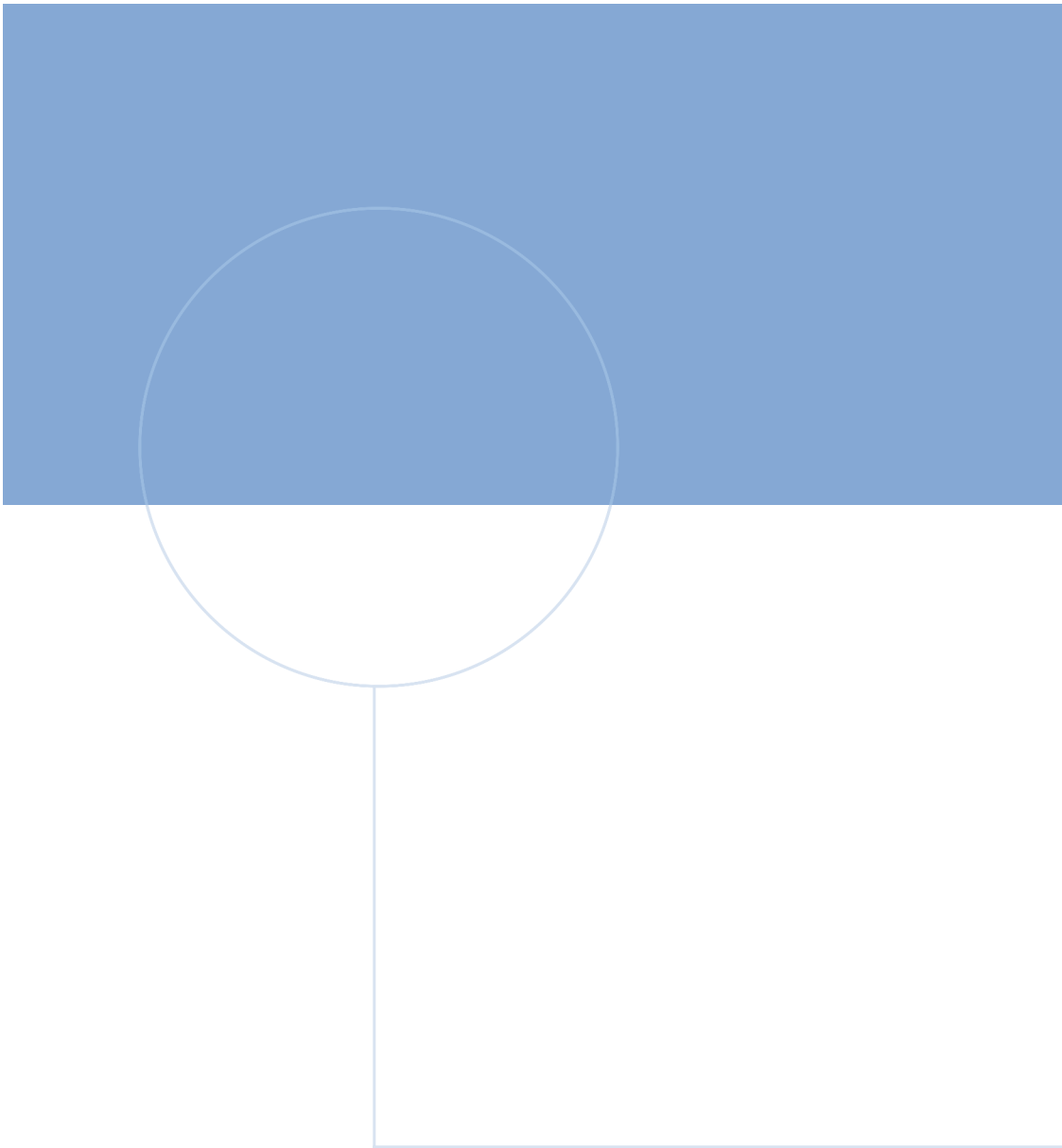
[49] Yu Zheng et al. “Mining interesting locations and travel sequences from GPS trajectories”. In: *Proceedings of the 18th international conference on World wide web*. 2009, pp. 791–800.

[50] Yu Zheng et al. “Understanding mobility based on GPS data”. In: *Proceedings of the 10th international conference on Ubiquitous computing*. 2008, pp. 312–321.

[51] Yu Zheng, Xing Xie, Wei-Ying Ma, et al. “GeoLife: A collaborative social networking service among user, location and trajectory.” In: *IEEE Data Eng. Bull.* 33.2 (2010), pp. 32–39.

[52] Sangdeok Park, Daesik Ko, and Seokil Song. “Parallel Insertion and Indexing Method for Large Amount of Spatiotemporal Data Using Dynamic Multilevel Grid Technique”. In: *Applied Sciences* 9.20 (2019), p. 4261.

[53] Ruichu Cai et al. “DITIR: distributed index for high throughput trajectory insertion and real-time temporal range query”. In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1865–1868.



NTNU

Norwegian University of
Science and Technology