TECHNISCHE UNIVERSITÄT BERLIN

A Database Benchmark for processing of 3D Spatio-Temporal Data

Master's Thesis

Author
Felix Bieleit-Medicus
373055
bieleit-medicus@campus.tu-berlin.de

Advisor
Tim Christian Rese

Examiners
Prof. Dr.-Ing. David Bermbach
Prof. Dr. habil. Odej Kao

Technische Universität Berlin, 2025
Fakultät Elektrotechnik und Informatik
Fachgebiet Scalable Software Systems

A Database Benchmark for processing of 3D Spatio-Temporal Data

Master's Thesis

Submitted by:
Felix Bieleit-Medicus
373055
bieleit-medicus@campus.tu-berlin.de

Technische Universität Berlin
Fakultät Elektrotechnik und Informatik
Fachgebiet Scalable Software Systems

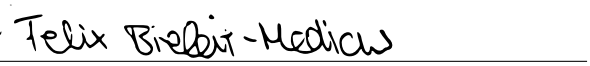
2025

Hiermit versichere ich, dass ich die vorliegende Arbeit eigenständig ohne Hilfe Dritter und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Alle Stellen die den benutzten Quellen und Hilfsmitteln unverändert oder sinngemäß entnommen sind, habe ich als solche kenntlich gemacht.

Sofern generische KI-Tools verwendet wurden, habe ich Produktnamen, Hersteller, die jeweils verwendete Softwareversion und die jeweiligen Einsatzzwecke (z. B. sprachliche Überprüfung und Verbesserung der Texte, systematische Recherche) benannt. Ich verantworte die Auswahl, die Übernahme und sämtliche Ergebnisse des von mir verwendeten KI-generierten Outputs vollumfänglich selbst.

Die Satzung zur Sicherung guter wissenschaftlicher Praxis an der TU Berlin vom 8. März 2017* habe ich zur Kenntnis genommen.

Ich erkläre weiterhin, dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.



(Unterschrift) Felix Bieleit-Medicus, Berlin, 5. März 2025

*https://www.static.tu.berlin/fileadmin/www/10000060/FSC/Promotion___Habilitation/Dokumente/Grundsaetze_gute_wissenschaftliche_Praxis_2017.pdf

Abstract

Spatiotemporal data, which combines both location and time information, is increasingly generated by IoT devices, GPS-enabled systems, and various tracking technologies. The sheer volume of this data requires scalable database systems capable of not only storing spatiotemporal information efficiently but also providing extensive querying capabilities. However, there is a lack of industry-scale databases that offer both efficient storage and comprehensive querying capabilities for managing spatiotemporal data effectively. MobilityDB aims to close this gap by offering abstract data types for representing continuously moving objects, along with a rich set of operators and functions for efficient data retrieval and analysis of spatiotemporal data. MongoDB is a widely adopted NoSQL database that supports geospatial data types and functions. However, it lacks the extensive spatiotemporal data types and functions provided by MobilityDB. In this thesis, we conduct performance and scalability benchmarks on MobilityDB and MongoDB, to see how they cope with spatiotemporal workloads. Our experiments include temporal, spatial, and spatiotemporal queries that can be implemented on both database systems, ensuring a fair comparison. We conduct the benchmarks using flight track data. Furthermore, to better reflect a real-world application scenario, we extend the benchmarks' queries to include regional data from the evaluated area.

The entire benchmarking setups, including the database systems and a custom benchmarking client, are deployed in cloud environments. Besides single-node setups, we also evaluate distributed setups of MobilityDB and MongoDB to assess how the databases cope with the ever increasing amount of spatiotemporal data. We find that MongoDB achieves lower latencies and higher throughput compared to MobilityDB in the single-node setup. However, latencies differ substantially across different query types for both database systems. In distributed setups, both systems demonstrate improved performance with lower latencies and higher throughput. Nevertheless, the performance gains are considerably higher for MongoDB compared to MobilityDB.

Kurzfassung

Raumzeitliche Daten, die sowohl Standort- als auch Zeitinformationen umfassen, werden zunehmend von IoT-Geräten, GPS-fähigen Systemen und verschiedenen Ortungstechnologien erzeugt. Die schiere Menge dieser Daten erfordert skalierbare Datenbanksysteme, die nicht nur in der Lage sind, raumzeitliche Informationen effizient zu speichern, sondern auch umfangreiche Abfragefunktionen bieten. Bislang gibt es jedoch keine industrietauglichen Datenbanken, die sowohl eine effiziente Speicherung als auch umfassende Abfragefunktionen für die effektive Verwaltung raumbezogener Daten bieten. MobilityDB zielt darauf ab diese Lücke schließen, indem es abstrakte Datentypen für die Darstellung von sich kontinuierlich bewegenden Objekten zusammen mit einer Vielzahl von Operatoren und Funktionen für die effiziente Datenabfrage und -analyse von raumzeitlichen Daten anbietet. MongoDB ist eine weit verbreitete NoSQL-Datenbank, die raumbezogene Datentypen und Funktionen unterstützt. Sie verfügt jedoch nicht über die umfangreichen raumzeitlichen Datentypen und Funktionen von MobilityDB. In dieser Arbeit führen wir Leistungs- und Skalierbarkeitsbenchmarks für MobilityDB und MongoDB durch, um zu sehen, wie sie mit raumzeitlichen Arbeitslasten zurechtkommen. Unsere Experimente umfassen zeitliche, räumliche und raumzeitliche Abfragen, die für beide Datenbanksysteme implementierbar sind, um einen fairen Vergleich zwischen den System zu gewährleisten. Wir führen die Benchmarks anhand von Flugdaten durch. Um ein realistische Anwendungsszenario besser abzubilden, erweitern wir die Abfragen der Benchmarks um regionale Daten aus dem untersuchten Gebiet.

Die gesamten Benchmark Konfigurationen, einschließlich der Datenbanksysteme und eines eigens implementieren Benchmarking-Clients, werden in Cloud-Umgebungen verwendet. Neben Setups mit einzelnen Datenbankknoten testen wir auch verteilte Setups von MobilityDB und MongoDB, um zu beurteilen, wie die Datenbanken mit der ständig wachsenden Menge an raumzeitlichen Daten zurechtkommen. Wir stellen fest, dass MongoDB im Vergleich zu MobilityDB in einer Konfiguration mit einem einzigen Datenbankknoten geringere Latenzen und einen höheren Abfragendurchsatz erreicht. Allerdings unterscheiden sich die Latenzen bei den verschiedenen Abfragetypen für beide Datenbanksysteme erheblich. Bei Konfiguration mit meheren Datenbankknoten zeigen beide Systeme eine höhere Leistung mit geringeren Latenzen und einem höherem Abfragendurchsatz. Allerdings sind die Leistungssteigerungen bei MongoDB im Vergleich zu MobilityDB deutlich höher.

Contents

1 Introduction

Devices which produce spatiotemporal data, i.e., data that has a location and a timestamp, are very common throughout the IoT and have led to the generation of vast amounts of spatiotemporal data [1][2]. While the problem of data acquisition is solved by the mass production of GPS chips and advances in automatic identification systems, there is a lack of tools for managing spatiotemporal data [3]. If spatiotemporal data describes moving objects, such as vehicles, consecutive data points can be interpolated to form trajectory data. Moving object databases (MODs) are built to manage space- and time-referenced objects and serve as a middle layer between the data and the applications handling it [3]. They introduce new spatiotemporal datatypes and offer operators and functions to handle these datatypes.

Several databases already offer geospatial support, such as the PostgreSQL extension PostGIS or the NoSQL database MongoDB, but spatiotemporal data is only rarely supported [4][5]. Benchmarks evaluating the performance capabilities of spatiotemporal databases are equally uncommon. Furthermore, in current benchmarks designed to evaluate the performance of spatiotemporal databases, the querying of the temporal aspect of the data objects is underrepresented [6]. Spatiotemporal databases must provide low latencies for real-time requests conducted on smaller data portions while also being capable of efficiently managing large data volumes for complex post-analysis queries. Without adequate benchmarks, researchers and developers lack information on which database best meets their needs.

To fill this gap, we aim to build an extensible benchmark to assess the performance of databases using spatiotemporal queries. The first database we benchmark is MobilityDB, which aims to be an industry-scale solution for managing spatiotemporal data [3]. MobilityDB aligns with the ongoing Open Geospatial Consortium (OGC) standards on Moving Features, which establish guidelines for accessing and managing moving features. These standards define methods for retrieving attributes and relationships between trajectory objects within a moving object database [3][7]. Current literature frequently uses the research prototype SECONDO for comparing the performance of spatiotemporal databases [3][8][9]. However, DISTRIBUTED SECONDO is no longer supported and relies on the outdated Apache Cassandra 3.0 as its storage layer. Therefore, we choose MongoDB as our second System under Test (SUT). Thus, we gain insights on the performance of spatiotemporal queries for a relational database system and for a widely adopted NoSQL database. We evaluate both single-node and three-node setups of these databases to assess their scalability and performance under increasing workloads. We populate the databases with data from the aviation industry and evaluate their performance using temporal, spatial, and spatiotemporal queries. Unlike MobilityDB, MongoDB does not natively support spatiotemporal data, as its support for spatiotemporal data types and queries is limited [4]. However, our goal is to employ a wide range of queries analyzing 3D spatiotemporal flight data, showcasing the strengths and weaknesses of both systems in managing spatiotemporal workloads.

When developing our benchmark, we adhere to general benchmark design objectives, such as reproducibility, relevance, and portability [10]. To assess the databases' performance, we use the latency and throughput of the queries. For each database, we implement a benchmarking client that issues queries while ensuring a sufficient load is generated on the SUTs. To guarantee an equal load on both systems and ensure comparability, we make

sure that queries retrieve identical results when executed with the same input parameters across both SUTs.

We aim to evaluate the databases under realistic conditions, such as when deployed in a cloud environment. However, benchmarks conducted in cloud environments can be highly variable, potentially leading to incorrect conclusions. Therefore, to obtain reliable measurements and ensure reproducibility, we repeat benchmark runs for both single-node and distributed setups across both SUTs.

To evaluate both systems under realistic conditions, we use detailed regional data alongside the dataset containing flight tracks over the German state of North Rhine-Westphalia. We conduct application-centric benchmarks in which we request the databases with temporal, spatial, and spatiotemporal queries in an arbitrarily order. Moreover, to assess the performances of the SUTs on different query types as well as specific queries, we conduct category benchmarks by executing spatial and spatiotemporal queries sequentially in separate runs on the SUTs.

In our work we aim to answer the following research questions:

- *What is the performance of state-of-the-art databases for handling spatiotemporal workloads?*
- *How to evaluate the scalability of databases for spatiotemporal queries using an application-centric workload?*

For single-node setups, we find that MobilityDB's latencies are, on average, 39% higher than those of MongoDB. However, the extent of this difference in latencies depends on the query type and on the specific query. While MongoDB demonstrates substantially lower latencies for temporal and spatial queries, its latencies for spatiotemporal queries are more comparable to those of MobilityDB.

For the distributed setups of MongoDB and MobilityDB, we find that MongoDB can handle a greater number of concurrent connections for executing queries compared to MobilityDB. MongoDB processes the set of queries substantially quicker than MobilityDB. MongoDB experiences greater performance gains from a three-node cluster deployment than MobilityDB, as indicated by its higher throughput. Nonetheless, both databases demonstrate notable improvements in performance with three-node configurations, exhibiting lower latencies and higher throughput compared to their single-node setups.

The work is structured as follows: first, we will present necessary background information related to benchmarking databases, spatiotemporal databases in general, and our SUTs, MobilityDB and MongoDB. Afterward, in Section 3, we outline the components used to evaluate the performance of the SUTs, the dataset for our experiments, the systems' setup processes, the experiment execution, and finally, the queries employed in the benchmarks. Consequently, we examine the results that we obtained in our benchmark runs. We discuss the obtained results and our benchmark approach in Section 5. Following that, we present other approaches to measuring the performance and scalability of databases using spatiotemporal workloads. Finally, in the conclusion, we summarize our work.

2 Background

2.1 Database Benchmarks

Benchmarking refers to the process of evaluating and comparing different components or entire IT systems according to specific qualities" [10]. Contrary to monitoring, benchmarking requires the practitioner to actively stress the SUT using a load generator component. Benchmarks should adhere to general design objectives, such as relevance, reproducibility, or fairness. Relevance prescribes that a benchmark pictures realistic scenarios, e.g., using queries for database testing that reflect requests made by an actual application. Reproducibility demands similar results when conducting the same benchmarks repeatably, which might pose a challenge when operating on shared resources with varying degrees of resource isolation, and thus varying performance. And finally, fairness prescribes that a benchmark is always domain-specific and does not overemphasize certain features [10].

IT benchmarks are subclassified into hard- and software benchmarks, aiming to answer a specific question. Moreover, there are application benchmarks and microbenchmarks. Application benchmarks evaluate deployed systems with all their related components in a production-like environment using an artificial but realistic workload. On the other hand, microbenchmarks evaluate the SUT by calling individual functions repeatedly with artificial parameter values. As they do not require the deployment of the entire system with all its components, they are easier to set up and execute but cannot detect all problems [11]. Benchmarks can measure a plethora of software qualities. These include non-functional requirements such as availability, security, elastic scalability, consistency, or performance. These qualities are never isolated, as the improvement of one quality is often accompanied by the deterioration of another. For our benchmarks of MobilityDB and MongoDB we evaluate the systems' performance and scalability.

As we benchmark a request/response-based system, performance has two dimensions, latency, and throughput. Latency measures the time between sending a request and receiving the response. Latency is always denoted by a positive value. Throughput expresses how many parallel request-response interactions a database is currently handling or is maximally able to handle and is usually described as requests per second. Although latency and throughput are dimensions of the same quality, they are contrary to each other. Databases optimize either for low latency or high throughput, as improving latencies come with an increased amount of threads competing for the same resources, thus decreasing throughput. Furthermore, there is a trade-off between latency and other qualities, such as security, i.e., confidentiality, as confidentiality requires encryption, which adds a latency overhead [10]. When observing the execution times, different sources of variability can influence the measurements at different granularities. To capture variability caused by factors that can occur at any time during the benchmark execution, benchmarks repeatedly execute the same task, in our case the same query but with different parameters, within a single process and measure the task execution time in each iteration. This decreases the impact of factors such as scheduling, memory caches, or background load. Furthermore, benchmarks are executed in multiple processes, i.e., benchmark runs, to capture factors that can change between runs.

The Yahoo! Cloud Serving Benchmark (YCSB) framework is a prominent suite for benchmarking NoSQL databases, which has the goal to facilitate the comparison of cloud data serving systems [12]. GeoYCSB extends YCSB to also be able to evaluate the performance and scalability of NoSQL databases for geospatial workloads [13]. They demonstrate their

framework by performing micro- and macrobenchmarks on performance and scalability on two leading document stores, namely MongoDB and Couchbase. In their result evaluation they emphasize the importance of search algorithm's efficiency of the spatial index structure on the query performance.

We utilize the cloud for running database benchmarks, as it enables us to distribute our databases across multiple machines efficiently. Furthermore, the cloud serves as an excellent testbed, allowing resource requirements to be easily adjusted to match the needs of the benchmark. However, benchmarks conducted in cloud environments can result in highly variable and unpredictable performance. Therefore, it is even more crucial to repeat benchmarks across multiple runs to ensure the results are reproducible.

2.2 Spatiotemporal Databases

Research in MODs has been active since the early 2000s, resulting in an extensive body of literature covering various aspects such as data modeling, operations, and indexing. However, despite this progress, only a few research prototypes exist, and a widely adopted, mainstream spatiotemporal database is still missing [3]. In this section, we present some of the existing systems that offer data management for moving object data.

Concrete and physical objects have a position and extent at any point in time. This applies to different objects such as countries, fishing vessels, people, or cars. For these space and time-referenced objects, also termed spatiotemporal objects, the past, current, and anticipated future position are of interest. This leads to the need for database systems to capture these aspects of objects [14]. Spatiotemporal objects can be categorized as either discretely moving or continuously moving. A discretely moving object, for example, could be a parcel progressing through a supply chain in discrete steps (e.g., from a warehouse to a truck to a distribution hub). Storing such objects, where locations change in discrete steps, can be achieved by using separate temporal and spatial columns in relational tables. However, it is less straightforward to store objects that change their position or extent continuously, such as cars or airplanes. In an early research, Güting et al. [14] propose to represent time-dependent geometries, such as moving points or regions, as attribute data types with appropriate operations. To achieve this, the database management system (DMBS) data model and query language should be extended with abstract data types. In addition to moving points and moving regions, several auxiliary data types are needed, such as a line type for projections or "moving reals" for time-dependent distances. When creating a powerful query language for spatiotemporal data Güting et al. emphasize the importance of consistent operations, i.e., operations should range over as many types as possible and behave consistently. The storage of continuously moving objects can be supported by a spatiotemporal database that provides abstract data types.

One of the main prototypes of spatiotemporal databases is SECONDO, an open-source DBMS that has been developed at the university of Hagen since 1995. It supports spatial and spatiotemporal datatypes [3][15]. SECONDO consists of three major components: a kernel, an optimizer, and a graphical user interface (GUI). The kernel is open for the implementation of a variety of DBMS data modules, and can be extended by algebra modules. SECONDO includes algebras for basic, spatial, and spatiotemporal data types. Queries are written in a procedural language, the so-called SECONDO executable language. SECONDO has been extended to DISTRIBUTED SECONDO to handle the rapidly growing spatiotemporal data volumes [16]. DISTRIBUTED SECONDO employs SECONDO for query processing and Apache Cassandra as a distributed data storage

system. Three node types are defined: *management nodes*, *storage nodes*, and *query processing nodes*. *Management nodes* run SECONDO processes that import and export data into and from DISTRIBUTED SECONDO. *Storage nodes* run Cassandra to store data, while *query processing nodes* and *management nodes* read and write data to these nodes. *Query processing nodes* are responsible for the computation of the queries. Their workload is primarily CPU-bound; therefore, *query processing nodes* can be deployed on the same machines as *storage nodes*, whose workload is I/O-bound. While previous experiments have demonstrated promising performance results in certain setups [3], DISTRIBUTED SECONDO is no longer supported and relies on the outdated Apache Cassandra version 3.0.

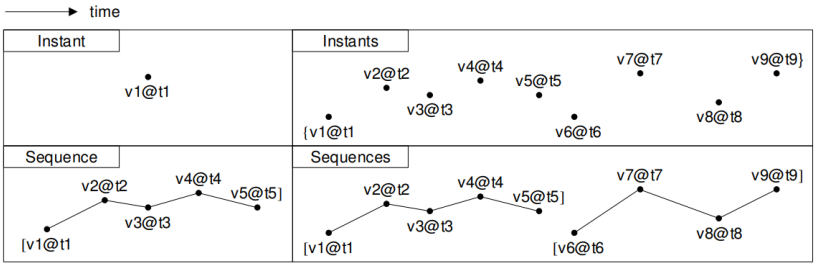
GeoMesa is a spatiotemporal data processing system that is built on top of NoSQL databases such as Cassandra, HBASE, or Accumulo. Efficient querying of spatiotemporal data is guaranteed by transforming multidimensional data (location, timestamp) into 1D keys using space-filling curves. In addition, GeoMesa supports optimized query performance through various indexing techniques on spatial and spatiotemporal data [17]. It leverages popular distributed databases to store its data, including geographic data such as points, lines, and polygons. GeoMesa is open-source and promises high read and write speeds on Big Spatial Data as well as horizontal scalability. Although claimed to be supported, GeoMesa still relies on the outdated Cassandra 3.0 for its data storage layer. Furthermore, the Common Query Language (CQL) used for querying data in GeoMesa supports only a limited set of operations [18]. It primarily supports filtering, while more complex operations, such as joins, are not supported. For these reasons, GeoMesa cannot currently be considered a full-fledged spatiotemporal database suitable for spatiotemporal analytics. However, GeoMesa allows for the integration into a stack of Accumulo, GeoMesa, Spark, and the Jupyter interactive notebook application. Using this stack allows for SQL queries to be performed on the spatiotemporal data [18].

TrajMesa is a distributed NoSQL trajectory storage engine built on top of GeoMesa. It extends GeoMesa's capabilities by enabling direct management of trajectory data [19] and supports various trajectory queries, including temporal queries, spatial range queries, and k-Nearest Neighbor (k-NN) queries. Additionally, TrajMesa incorporates standard preprocessing, indexing, and storage processes. Each row in a TrajMesa table contains the corresponding trajectory, its spatiotemporal properties—such as the trajectory's minimum bounding rectangle (MBR) and time span—and a signature describing its path. The sequence of GPS coordinates forming the trajectory is compressed using GZip to reduce storage costs [19].

2.3 MobilityDB

PostgreSQL is an open-source object relational database management system (RDBMS). CRUD operations on the relational tables of a PostgreSQL system can be done via standard SQL [20]. PostGIS is an extension adding comprehensive geo-functionalities with more than 1000 spatial functions to PostgreSQL. Its implementation is based on "lightweight" geometries and comes with optimized indexes to reduce disk and memory usage [21]. A wide range of geometry and geography types are supported, such as *Points*, *LineStrings*, *Polygons*, or *Multipolygons*/*MultiLineStrings*. The choice between the geometry and geography type depends on the area covered by the geospatial data and by the queries. Geometry types store spatial data using planar coordinate systems, making them ideal for small-scale or localized queries where the Earth's curvature is negligible. In contrast, geography types use a spherical coordinate system, ensuring accurate distance and area

Figure 2.1: Type constructors of MobilityDB [3]



calculations over large-scale or global queries by accounting for the Earth's curvature [3]. With the PostGIS extensions geospatial measurements like area, distance, length, or perimeter can be determined in PostgreSQL. Furthermore, for high-speed spatial querying, GiST (Generalized Search Tree) indexes can be used on geometric or geographic data types.

MobilityDB is an open-source extension for PostgreSQL that builds on PostGIS to support moving objects. It introduces temporal and spatiotemporal abstract data types (ADTs) that leverage PostgreSQL and PostGIS type systems to represent moving objects [3]. These ADTs are fully integrated into PostgreSQL and PostGIS, allowing MobilityDB to benefit from the continuous development of the underlying platform. MobilityDB introduces *temporal types* that represent the evolution in time of another type, called the *base type*. This *base type* can be `text`, `int`, `boolean`, `float`, `geometry`, or `geography`. Consequently, the *temporal types* are: `ttext`, `tint`, `tboolean`, `tfloat`, `tgeometry`, and `tgeography`. The temporal types `tgeometry` and `tgeography` directly build on the `geometry` and `geography` types introduced by PostGIS. The *temporal types* can be used to capture changing states over time, such as variations in temperature with `tfloat` or whether a cab is occupied with `tboolean` [3]. The data types for moving objects can be constructed using the four type constructors `INSTANT`, `INSTANTS`, `SEQUENCE`, and `SEQUENCES`. For instance, `INSTANTS(geometry)` can represent changing locations of a user logging into a location-aware application. `SEQUENCE(geometry)`, on the other hand, implies linear interpolation between consecutive time instants and therefore can represent continuous trajectories such as a car ride [22]. The four type constructors are illustrated by Figure 2.1, in which `v@t` denotes the value of the *base type* `v` that occurs at the time instant `t`.

The *temporal types* are supported by the generalized search tree (GiST) index, the space partitioned search tree (SP-GiST) index, and the B-tree index. All three indexes are generic. The GiST and SP-GiST indexes support spatiotemporal, spatial-only, and temporal-only queries. The B-Tree index supports equality and range queries on sortable data and is primarily used in MobilityDB for internal query sorting. The GiST index acts as a template for implementing various indexing schemes and implements an balanced R-Tree over the indexed type. The SP-GiST index supports unbalanced trees and also acts as a template for various indexing schemes [3]. In MobilityDB, bounding boxes and periods are stored for the GiST and SP-GiST indexes, allowing them to accelerate queries that involve bounding box predicates. This is demonstrated later in Section 3.

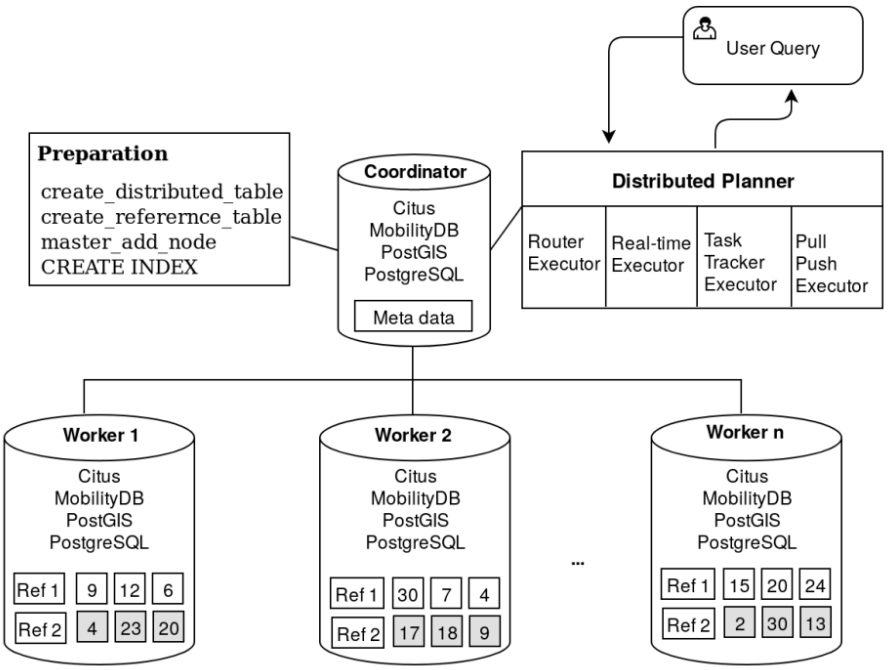
MobilityDB enhances the relational database with a rich set of query operations for managing spatiotemporal trajectory data [23][22]. It implements the essential technical foundation, including constructors, casts, and functions specified or recommended by ISO

and OGC[1] standards on moving features [3]. Because MobilityDB's operations are polymorphic, the resulting type depends on the input types. For instance, many operations that accept `tgeompoint` as an input type also support `tgeogpoint`. A significant portion of MobilityDB's temporal operations is derived through *lifting*, a process that extends spatial operations from PostGIS to work with temporal data types [22][3]. In line with the proposal by Güting et al. [14], the newly introduced operations include *lifted* topological predicates from PostGIS, such as `tintersects`, which identifies the points in time when a `tgeompoint` or `tgeogpoint` (collectively referred to as `tpoint`) intersects a `geometry` or `geography`. Additionally, the system supports simple topological predicates like `adwithin`, which checks whether two `tpoint` trips remain within a specified maximum distance at all times. Distance operators further enhance MobilityDB's functionality by providing various ways to measure proximity between objects, whether both are in motion or one is stationary. For instance, `nearestApproachInstant` returns the `instant` at which two moving objects of type `tpoint`, or a moving object of type `tpoint` and a static object of type `geometry`/`geography`, are closest to each other [24][3]. MobilityDB also supports spatiotemporal functions. For instance, the function `speed(tpoint)` computes a `tfloat` representing an object's changing speed over time, while the OGC-standard function `cumulativeLength(tpoint)` returns a `tfloat` that represents the cumulative distance traveled as a function of time. These examples represent only a subset of the extensive functionality available in MobilityDB. A comprehensive list of supported operations can be found in the official MobilityDB documentation[2].

Citus is an extension for PostgreSQL that enables horizontal scalability for the relational database. It can be combined with MobilityDB to perform distributed spatiotemporal query processing and handle the growing volume of moving object data [23][22]. Citus supports both sharding and data replication across multiple machines [9]. Figure 2.2 illustrates the overall architecture of MobilityDB integrated with the Citus extension. All machines in the cluster run the same technology stack, which consists of PostgreSQL, PostGIS, MobilityDB, and Citus. One node acts as the *coordinator*, managing client interactions and distributing queries to worker nodes. It does this by consulting the metadata tables. These are stored on the coordinator node and track the health of workers, and distribution of data across the nodes [25]. The `create_distributed_table` command creates shards that contain table rows and distributes them across multiple workers, thereby reducing the amount of data stored on a single node. It achieves this by sharding based on a table column, known as the `sharding key`. In contrast, `create_reference_table` replicates tables so that an entire copy of the data is available on each node, including the *coordinator*. This can be beneficial for smaller tables that are frequently accessed and involved in join queries, speeding up query performance. When an index is created on a distributed table, the coordinator pushes the `CREATE INDEX` statement down to the individual shards. Each worker then builds a local index and uses it to optimize the parts of queries it receives from the coordinator [23][22]. For incoming queries the *coordinator* generates a distributed query plan consisting of partitioned smaller query fragments that can be executed independently by the workers holding the shards. After the *worker* nodes execute their tasks, they send the results to the *coordinator*, which merges them and delivers the final result to the client. To guarantee fault tolerance, the *coordinator* is constantly monitoring the workers, and re-assigns the query task to another worker in case a worker is crashing during query execution. Citus distinguishes four different types of queries: *routable* queries, *push downable* queries, *recursive CTE* queries, and *complex*

[1] `https://www.ogc.org/de/publications/standard/movingfeatures/`
[2] `https://mobilitydb.github.io/MobilityDB/master/`

Figure 2.2: Distributed MobilityDB architecture [9]



queries. *Routable* queries must include a constraint based on the sharding key. This ensures that the query is directed only to the worker nodes containing the relevant shards and, consequently, the necessary data for the specified key value. Routing queries for distributed MobilityDB using the Citus extension can be utilized for, e.g., multi-tenant applications. For instance, if the dataset contains trajectories of a transport company's fleet in different cities, the sharding key can be set to the city name. Consequently, vehicle data for each city would be assigned to a specific shard, allowing queries targeting that city to be routed to the *worker* node storing the corresponding shard [22]. *Push downable* queries span multiple shards and are distributed in a single round to the *worker* nodes. Workers locally optimize the execution of their fragments and send their individual results back to the coordinator [22]. `Recursive CTE` queries recursively call the planner for subqueries. The results are then pushed back to the workers, which use them as intermediate results for evaluating the main query. By default, the Citus planner rejects non-co-located joins, as they require costly network I/O for re-partitioning the data [22].

2.4 MongoDB

The exponential growth of data has driven the emergence of NoSQL databases, as relational databases like PostgreSQL can experience performance degradation when handling massive volumes of information [26]. NoSQL, which stands for "Not only SQL", broadly refers to a class of non-relational data stores. These systems are widely adopted in modern applications due to their scalability and high performance [4]. One of the main differences to relational databases is that NoSQL datastores can handle unstructured data, such as documents, e-mail, or multimedia. NoSQL databases follow a very simple data model and a flexible schema [27]. There are four major categories of NoSQL databases: Key-Value databases, document based databases, columnar databases, and graph-oriented databases. MongoDB is a scalable document based database with a large user base, which was originally launched in 2009 [27]. MongoDB supports similar features as SQL-based databases

such as sorting, indexing, range querying, and and updates [21]. Stages of these operations can be combined into aggregation pipeline to build more complex queries for MongoDB. Furthermore, MongoDB supports horizontal partitioning of large data volumes, distributing data across multiple servers within a cluster. This horizontal scaling allows the system to handle larger datasets efficiently without degrading performance, even when individual servers reach their capacity [13]. MongoDB databases are comprised of a set of collections, which are similar to relational databases' tables but do not require a predefined schema. Collections store data in BSON documents, which are binary-encoded, JSON-like objects. Collections store individual observations in documents, and each document contains fields that store the attributes of those observations [27]. Fields can contain basic data types, or more complex structures such as lists or even documents itself.

MongoDB provides spatial functionality by storing data in the GeoJSON format[3]. This encoding supports various geographical data structures, including `Points`, `LineStrings`, and `Polygons`. A GeoJSON document consists of one field specifying the GeoJSON type and another field containing the coordinates of the geographical structure [5]. So far, MongoDB supports queries for geographical containment, intersection, and sorting by distance [20]. Geographic information must be stored in the GeoJSON data structure which restricts the schema-less approach of MongoDB. However, this allows the geospatial indexes, 2d and 2dsphere, to be built on GeoJSON fields. These geospatial indexes differ in their underlying calculations: the 2d-index operates on a flat, two-dimensional plane, making it suitable for euclidean distance calculations, whereas the 2dsphere-index models data on an Earth-like sphere, enabling accurate geospatial queries that consider the Earth's curvature. The 2dsphere-index supports all of MongoDB's geospatial queries, `$geoIntersects`, `$geoWithin`, `$geoNear`, `$near`, and `$nearSphere`, and operates on the WGS 84 reference system [21]. Some of the the geospatial queries come with limitations; for instance, the `$geoNear` query can only be used in the first stage of an aggregation pipeline. To efficiently store spatial data, MongoDB utilizes GeoHashes to map coordinate pairs to one-dimensional values [4]. These geohashed values can then indexed to speed up query processing [21]. The 2dsphere-index partitions the Earth's surface into hierarchical grid cells at multiple resolution levels. Each of these cells is then indexed using the B^+-tree [28]. B^+-trees order data linearly and provide a tree structure on it for faster retrieval [29]. MongoDB supports indexes on single fields, and embedded fields. Furthermore, MongoDB supports compound indexes, which are particularly useful when applications repeatedly run queries that contain multiple fields. For compound indexes data is grouped by the first field and then by each subsequent field [30].

MongoDB supports storing time series data in time series collections. Time series data typically consists of a timestamp, indicating when the data point was recorded; metadata, which rarely changes and identifies a data series; and measurements, representing data points tracked at increments in time. MongoDB automatically groups documents of time series collections into buckets that have an identical `metafield` value and have `timeField` values that are close together. Time series collections utilize a columnar storage format and store data in time-order, which can lead to improved query efficiency and reduced disk usage [30].

For distributed setups MongoDB supports two techniques, replication and sharding. Replication is implemented through *replica sets*, which are groups of MongoDB processes that maintain the same data. Therefore, data is redundant, highly available, and in case single

[3] `https://geojson.org/`

servers crash, not lost [30]. Sharding, on the other hand, distributes data across multiple machines, reducing the amount of data stored on each machine. Therefore, database systems handling large datasets and high-throughput workloads that exceed the capacity of a single server can distribute the load across multiple machines. Documents in collections are distributed based on their shard key, which consists of one or more fields from the document [30]. MongoDB partitions sharded data in chunks. MongoDB supports both *hash* and *range* sharding. Range sharding increases the likelihood that documents with similar shard keys are assigned to the same chunk or shard, whereas hash sharding ensures an even distribution of data. Sharding requires an index to be built on the shard key [5]. Sharded clusters are made of the following components: *shard servers*, *routers*, and *config servers*. *Shard servers* contain a subset of the sharded data and are deployed as replica sets. The *router* instances act intermediaries between the client and the database and direct queries to the appropriate *shard servers*. *Config servers* store metadata and configuration settings for the sharded cluster and should be replicated across all machines in the cluster to ensure fault tolerance.

Alongside MobilityDB, we use MongoDB as the second SUT for the spatiotemporal benchmarks, as most NoSQL databases currently lack support for spatial functions. Furthermore, as open-source software, it allows for the addition of more temporal, spatial, and spatiotemporal functions in the future. Additionally, it supports distributed setups, which are crucial for handling the large volumes of spatial data generated by modern systems [20][21]. However, unlike MobilityDB, MongoDB does not support abstract data types or linear interpolation, which are crucial for storing continuously moving object data.

3 Benchmark Design

In this chapter, we present the benchmark design, detailing the benchmarking architecture, the dataset used, and the system setup for both single-node and distributed configurations. We also provide an overview of the queries used to evaluate the performance and scalability of the SUTs. Additionally, we explain the how we conduct the various benchmark runs. The entire implementation is available on GitHub[4].

3.1 Benchmarking Architecture

Our benchmarking architecture comprises two main components: the *Benchmarking Manager* and the *Benchmarking Client*. The *Benchmarking Manager* is responsible for orchestrating both the benchmark preparation and execution phases by coordinating with the *Benchmarking Client*. While the *Benchmarking Manager* remains identical for both SUTs, we have implemented two versions of the *Benchmarking Client* component—one for benchmarking MobilityDB and another for benchmarking MongoDB.

Queries, along with their execution count and parameters, are specified in YAML configuration files. The benchmark framework is easily extendable to accommodate new queries by simply adding them to the YAML file. However, for MongoDB, while queries are listed in the YAML file, they are not directly implemented there. Instead, corresponding functions must be added to the MongoDB *Benchmarking Client* to ensure proper execution.

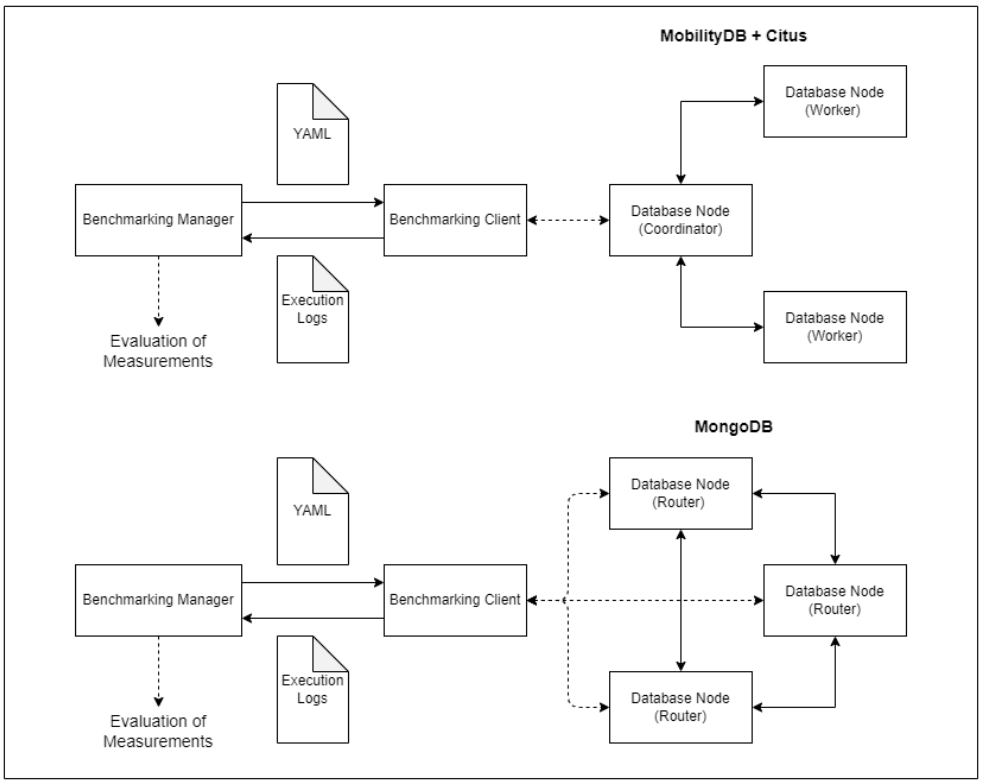
The *Benchmarking Manager* communicates with the *Benchmarking Client* via HTTP. During the preparation phase, the *Benchmarking Client* ingests the data into the databases and performs the necessary benchmark preparation steps, which are detailed in the subsequent section.

During the benchmark execution phase, the *Benchmarking Client* injects pseudo-random values into the queries, based on the parameters that are specified for each query in the YAML configuration files. By varying these parameter values, we ensure that the databases do not cache responses. The client adjusts parameter values for queries when functions of MongoDB and MobilityDB require different units. For example, MongoDB's `$centerSphere()` function expects the radius in radians to compute a circular area around a point, whereas MobilityDB's functions, when working with points of type `geography`, require the radius to be specified in meters for the corresponding computations. To ensure consistent results for the same queries across both SUTs and enable a fair comparison between the systems, the client appropriately converts the queries' parameter values. For inserting periods into the queries, the *Benchmarking Client* distinguishes between `period_short` (periods ranging from 0 to 2 days), `period_medium` (2 days to 15 days), `period_long` (15 days to 1 year), and the fully random `period` (0 to 1 year). These periods are defined with second-level precision rather than being limited to whole days. Furthermore, the periods always remain within the year 2023, which corresponds to the time-frame of the dataset we use. The temporal selectivity of a query can be modified by adjusting the period type that is set as an parameter for that query.

After injecting parameter values into the queries, the client executes them against the SUT. The Benchmarking Client records the start time for each query and captures the end time upon receiving the response. It then logs the query name, query type, parameter

[4] `https://github.com/FelixMedicusf/master_project`

Figure 3.3: Benchmarking setups for distributed deployments of MobilityDB and MongoDB



values, start time, and end time to a file. The load on the SUT can be adjusted by modifying the number of client threads that are executing queries.

For testing purposes, the responses of the database systems can be logged on the *Benchmarking Client*. Logging of the databases' responses can be configured via the `benchmark settings` in the YAML files, ensuring that both MobilityDB and MongoDB are subjected to the same workload and produce identical responses.

After executing the queries, the *Benchmarking Manager* retrieves the benchmark execution logs from the *Benchmarking Client* for further analysis. Figure 3.3 shows the components for benchmarking distributed setups of MobilityDB and MongoDB.

3.2 Data

For the benchmark, we use aviation industry data provided by Deutsche Flug Sicherung (DFS). The dataset includes flight tracks over North Rhine-Westphalia (NRW), Germany, covering altitudes from 0 to 66,000 feet. Since the dataset only covers flights within the airspace of North Rhine-Westphalia, any flight that exits and later re-enters this airspace is recorded as separate tracks. The dataset's variables are listed in Table 3.1.

Before inserting the tuples into our databases for the experiment, we preprocess the dataset by discarding flights with missing fields or corrupted data. Furthermore, some flights, identified by a unique flightId, had overlapping time ranges across different tracks, which should not be possible. To resolve this, we constructed new flights for those by modifying

their flightIds. For storing locations in the databases, we use the World Geodetic System 1984 (WGS84) as the Spatial Reference System due to its widespread use in the Global Positioning System (GPS)[5] and global mapping applications, including Google Maps[6]. Consequently, we transform the dataset's raw coordinates, originally in UTM format, into degrees of longitude and latitude.

Table 3.1: Fields in the aviation dataset provided by the DFS

Field Name	Description
`FlightID`	Flight ID. Unique identifier for a flight.
`AircraftType`	Type of aircraft.
`OriginAirport`	Departure airport (in ICAO format).
`DestinationAirport`	Destination airport (in ICAO format).
`NumberOfTracks`	Number of tracks belonging to a flight.
`TrackIdent`	Track number.
`TrackStartTime`	Time of the first track point within the evaluated area (UTC).
`TrackEndTime`	Time of the last track point within the evaluated area (UTC).
`TrackNoOfPoints`	Number of points in the track, truncated to fit the evaluated area.
`TrackPoint`	Point where a coordinate is recorded. 4-second intervals used.
`UTMCoordinates`	Coordinates in UTM format.
`Altitude`	Altitude in feet.

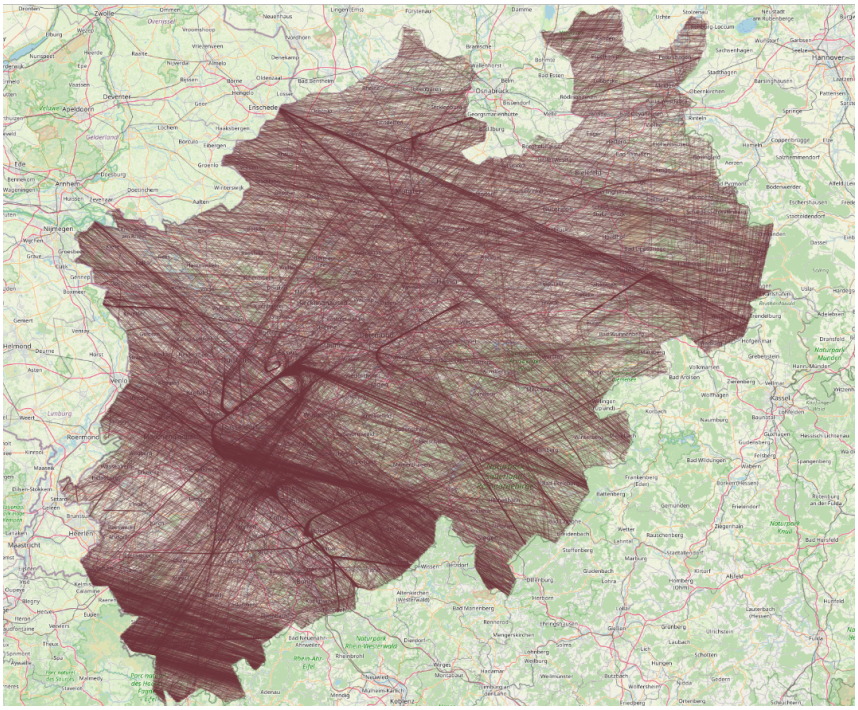
After preprocessing the DFS data, we obtain 163,269,637 tuples, each representing a flight track's position at a specific moment in time, recorded at four-second intervals. Each record includes the flight track's metadata such as the `flightID`, `track`, `aircraftType`, `originAirport`, and `destinationAirport`. Meanwhile, the aircraft's `altitude` and position, composed of `longitude` and `latitude`, dynamically change over time, reflecting the flight's movement. In total, these 163,269,637 tuples constitute the trip data for approximately 1.8 million flight tracks. Figure 3.4 presents a subset of these flight tracks.

MobilityDB requires to set an interpolation strategy for the `tpoint` type since its base type is continuous [3]. To ensure a fair comparison between both systems, we opt for stepwise interpolation, as linear interpolation is not supported by MongoDB, as it is not a native MOD. However, linear interpolation would more accurately reflect the continuous movement of the aircraft. To enhance temporal resolution, we apply interpolation to each flight track, gradually estimating the aircraft's position at intermediate seconds between recorded data points. This approach ensures a smooth transition between positions, providing a more continuous representation of the flight's three-dimensional trajectory over time. Although MobilityDB's `tpoint` datatype natively supports both stepwise and linear interpolation, we chose to perform the interpolation beforehand, prior to executing the benchmarks. This ensures that queries requiring interpolation—such as those retrieving positions for timestamps between observed aircraft locations—do not rely on real-time interpolation on the database side. By precomputing interpolated values, we reduce the computational overhead during query execution, thereby ensuring fairness in the comparison of the two SUTs. Interpolating the flight points increases the total number of data points from 163,269,637 to 647,462,528, while maintaining approximately 1.8 million flight tracks. Furthermore, for both databases, we constructed a *LineString* for each flight

[5] `https://epsg.io/4326`

[6] `https://developers.google.com/maps/documentation/javascript/coordinates`

Figure 3.4: Subset of the flight tracks in 2023 over North Rhine-Westphalia



track to effectively represent the continuous two-dimensional trajectory of a flight. This structure is used for spatial queries in our benchmarks.

To implement a realistic benchmark, we enrich our SUTs with comprehensive geographical data from NRW. This dataset includes:

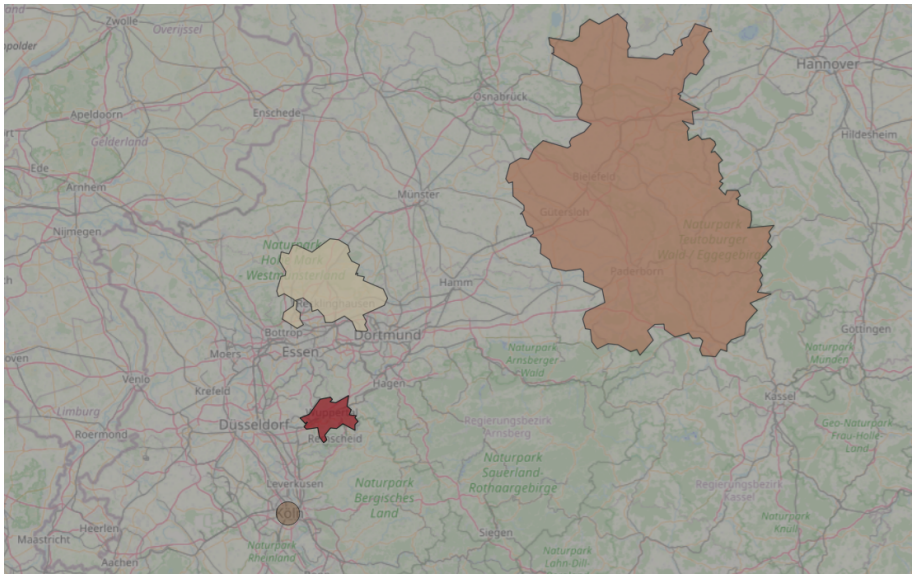
- **396 municipalities**,
- **53 counties**,
- **5 districts**, and
- **272 cities**

Each municipality, county, and district is represented as a polygon, defining its spatial extent, while cities are represented as point locations. We source the geographical data from Geobasis NRW[7]. Figure 3.5 presents representative examples of polygonal representations for a municipality, county, and district, along with a radius around a city, which were used in queries for our benchmarks. Geobasis NRW provides high-resolution polygons for municipalities, counties, and districts, with each polygon consisting of thousands of points. Since this substantially impacts the performance of spatial functions on these polygons and such high resolution is unnecessary, we optimize the polygons by reducing the number of points. To achieve this, we apply the Douglas-Peucker[8] algorithm to simplify the polygons before inserting them into the databases.

[7] `https://www.bezreg-koeln.nrw.de/geobasis-nrw/produkte-und-dienste/verwaltungskarten-und-grenzen/digitale-verwaltungsgrenzen`

[8] `https://cartography-playground.gitlab.io/playgrounds/douglas-peucker-algorithm/`

Figure 3.5: Exemplarily extent of different regions within NRW



From bottom left to top right: a radius around the city Cologne, the municipality of Wuppertal, the county of Recklinghausen, and the district of Detmold.

Additionally, we insert airport data into the databases to enhance the relevance of queries involving flight origin and destination information. This allows for more meaningful analysis of the historical flight data.

3.3 MongoDB

When setting up MongoDB, we face the challenge of designing an appropriate data model. Two primary approaches emerge:

(a) Storing each flight track as a single document, containing metadata such as `flightID`, `track`, `aircraftType`, `destinationAirport`, and `originAirport`, while embedding dynamic data—`timestamp`, `location`, and `altitude`—as nested documents within an array [5].

(b) Storing each flight data point as a separate document.

We opted for the latter based on preliminary tests, which revealed that queries on this model perform substantially better than those on option **(a)**. The array-based structure in option **(a)** requires potentially costly `$unwind`[9] stages, which may negatively impact query performance. Furthermore, we choose to store the flight data points in a time-series collection, as this approach offers improved query efficiency, reduced disk usage, and lower I/O for read operations [30]. The time-series collection contains the interpolated data points, totaling 647,462,528 documents. Figure 3.6 showcases the data modeling format.

We create an additional collection to store flight track metadata along with their trajectories, represented as GeoJSON objects of type *LineString*. This collection is used for spatial `$geoIntersects` queries on the trajectory field.

[9] `https://www.dragonflydb.io/faq/mongodb-unwind-slow-performance`

Figure 3.6: Exemplary flight point document of the time-series collection

```
{
  "_id": { "$oid": "67a0e2df3504f97bf2f8438e" },
  "metadata": {
        "airplaneType": "B738", "destinationAirport": "EDDL",
        "flightId": 693811770, "originAirport": "GCFV", "track": 1
        },
  "timestamp": { "$date": "2023-01-16T21:33:14.000Z" }
  "altitude": 904.8,
  "location": {
        "type": "Point",
        "coordinates": [6.899726, 51.3497383]
        },
}
```

In addition to the aforementioned collections, the MongoDB database also includes the `flightpoints` collection, which stores the raw, non-interpolated flight data points.

Furthermore, we create B-tree indexes on the altitude and timestamp fields, along with hashed indexes for fields used in join operations within our queries. To accelerate spatial queries, we also create `2dsphere` indexes on all GeoJSON fields. Efficient querying of spatiotemporal data can be achieved using a compound index. In our case, there are two possible configurations for a spatiotemporal index, differing in the order of fields: `(location, timestamp)` and `(timestamp, location)`. Since most of our queries exhibit high selectivity in the temporal dimension while imposing a less restrictive constraint on the spatial dimension, we opt for a compound index on `(timestamp, location)` [4].

MongoDB does not support shard keys based on a `2dsphere` index. Therefore, for our experiments on a distributed MongoDB cluster, we shard the time-series collection using a single-field B-tree index on the `timestamp` field. This approach provides a key advantage: temporal and spatiotemporal queries are routed only to the shards containing documents within the query's temporal constraints, minimizing unnecessary query distribution across the cluster.
To ensure an even distribution of data, we create a number of chunks equal to the number of shard servers in the cluster, with each chunk covering approximately $\frac{1}{\#\text{shard servers}}$ of the year 2023. In our setup, we assign the three resulting large chunks to the three *shard servers*, ensuring that most temporal and spatiotemporal queries targeted a single shard rather than spanning multiple shards. As a result, queries are efficiently routed to relevant shards instead of being broadcast across the entire cluster [4].
Additionally, we structure the chunk time ranges to cover approximately the same number of documents, ensuring load balancing across the cluster. That means that the chunk comprising the summer months covers slightly less than one-third of the year, as flight traffic is typically higher during that time. To further optimize data distribution, we enable MongoDB's *load balancer*, which automatically split and migrates chunks in case imbalances arise, redistributing data to *shard servers* with lower storage loads.

3.4 MobilityDB

In MobilityDB, we store each flight track as a separate row in the `flights` table, resulting in approximately 1.8 million rows. To achieve this, we utilize the abstract data types provided by MobilityDB. While the temporal data type `tfloat` stores the aircraft's altitude over time, we use the `tgeogpoint` type in the `trip` column to store its changing two-

dimensional location. We build the `altitude` and `trip` attributes using the `SEQUENCE()` type constructor on the interpolated flight data points. The `traj` column is of type `geography`, a datatype introduced by PostGIS, and stores the flight track's trajectory as a *LineString*. All other columns utilize standard PostgreSQL data types. For the temporal types of the columns `altitude` and `trip`, we apply the step interpolation. Table 3.2 presents the structure of the `flights` table.

Equivalently to MongoDB, the table `flightpoints` contains the flight data points before interpolation, thus 163,269,637 rows. The observed aircraft's locations are stored in the column `location` of type `GEOGRAPHY`.

We construct GiST indexes on the `altitude`, `trip`, and `traj` columns of the `flights` table, as well as on the `location` column of the `flightpoints` table. Additionally, we build a B-tree index on the `altitude` column of the `flightpoints` table. Furthermore, we create hash indexes on all columns involved in join operations across our queries.

Column Name	**Data Type**
`flightId`	integer
`track`	integer
`aircraftType`	text
`origin`	text
`destination`	text
`altitude`	tfloat
`trip`	tgeogpoint
`traj`	geography

Table 3.2: Structure of the Flights Table in MobilityDB

We are scaling MobilityDB horizontally using the PostgreSQL extension Citus[10]. The distributed planners and executors of Citus do not understand the MobilityDB types and operators. However, MobilityDB operations are built using the extensibility features of PostgreSQL. Therefore, many of MobilityDB's operations can be distributed, although Citus is not aware of their semantics [22]. Bakli et al. examine how different partitioning methods affect query performance on distributed setups of MobilityDB. They evaluate object-based partitioning, 3D grid partitioning, and GiST partitioning. However, since the utilized distributed queries all belong to the push downable class, and those are only affected by distribution balance, they find the performance difference of the examined methods to be small. Similarly, since our queries also fall into the push-downable class, we choose object-based partitioning for distributing the flight data, i.e., the tables `flights` and `flightpoints`. Furthermore, we opt for object-based partitioning due to its simplicity of implementation. We shard the `flightpoints` using the hashed `timestamp` column and the `flights` table using the hashed `flightId` column. For both tables, we set the `shardCount` to 16, ensuring that each *worker* node holds 8 shards and thereby achieve a well-balanced data distribution.

3.5 Queries

When designing our queries, we aim to cover a wide range of functions and application domains. Moreover, we want to use queries that reflect real-life scenarios. Similar to

[10] `https://www.citusdata.com/`

BerlinMOD [31] we employ queries of the temporal, spatial, and spatiotemporal type.By incorporating a balanced mix of temporal, spatial, and spatiotemporal queries, we avoid overemphasizing specific database features, ensuring a fair comparison. These queries can be applied in various contexts, including air traffic management and optimization, aviation safety and incident analysis, environmental and noise pollution monitoring, as well as urban and regional planning.

Compared to MobilityDB, MongoDB provides relatively limited support for temporal, spatial, and spatiotemporal operators and functions. Therefore, for our experiments we only employ queries that can be implemented for both SUTs. For instance, MongoDB does not support distance calculations between moving objects, thus, we do not include queries measuring the distance between those. Implementing such functionality would require custom-built functions for MongoDB. However, these types of queries could be valuable for various domains, such as air space surveillance or incident analysis [6].

To ensure fairness in the comparison, we verify that both SUTs return the same data when queried with identical parameters. The colon (:) annotation represents placeholders for parameters, which are dynamically assigned values by the Benchmarking Client. MobilityDB incorporates a set of Boolean predicates that are available for temporal types. The operators consider the topological relationships between bounding boxes, such as the overlaps operator (`&&`), which checks whether the bounding boxes of its two arguments have a non-empty overlap [22]. These operators consider values and/or dimensions based on the argument types. When running queries against MongoDB, we ensure that the initial stages of the aggregation pipeline filter documents by their timestamps. This allows queries to be efficiently routed to the appropriate *shard servers* storing the relevant documents.

Now, we present the queries we implemented for MobilityDB, which were equivalently implemented for MongoDB. For both databases, we experimented with different query variations and fine-tuned their performance.

Query 1 Retrieve the number of flights that occurred during the specified period.

```
SELECT COUNT(*), :period_medium
FROM flights f
WHERE f.trip && :period_medium;
```

This query can serve for basic air space traffic analysis and spot peak hours in North Rhine-Westphalia by identifying time periods with the highest flight activity. The overlaps operator (`&&`) is applied to the temporal dimension of the `trip` column, which is of type `tgeogpoint`, to filter out flights whose time spans do not overlap with the specified period. The GiST index on `trip`, of which only the temporal dimension is traversed, speeds up this query [3].

temporal - range - aggregation

Query 2 Retrieve the altitude and location of a flight at a specified instant.

```
SELECT f.flightid, valueAtTimeStamp(f.altitude, :instant),
       ST_asText(valueAtTimeStamp(f.trip, :instant)) AS location, :instant
FROM flights f
WHERE f.trip && span(:instant);
```

By capturing a flight's altitude and location at a specific instant, this query enables retrospective analysis of air traffic patterns. The predicate `f.trip && span(:instant)` evaluates whether the bounding box of `:instant` and the bounding box of `f.trip` overlap and triggers the use of the spatiotemporal GiST index which we build on the `trip` column. Without this predicate the query optimizer is left without hints, substantially increasing the query's execution time [3].

temporal - point - no aggregation

Query 3 Compute airport traffic based on departures and arrivals.

```
WITH departures AS (
    SELECT f.origin AS airport,
    COUNT(DISTINCT f.flightid) AS departure_count
    FROM flights f
    WHERE f.trip && :period_short
    GROUP BY f.origin),
arrivals AS (
    SELECT f.destination AS airport,
    COUNT(DISTINCT f.flightid) AS arrival_count
    FROM flights f
    WHERE f.trip && :period_short
    GROUP BY f.destination)
SELECT COALESCE(d.airport, a.airport) AS airport,
       COALESCE(d.departure_count, 0) AS departures,
       COALESCE(a.arrival_count, 0) AS arrivals,
       COALESCE(d.departure_count, 0) +
       COALESCE(a.arrival_count, 0) AS traffic_count
FROM departures d
FULL JOIN arrivals a ON d.airport = a.airport
ORDER BY traffic_count DESC, departures DESC, arrivals DESC;
```

This join query provides an overview of airport utilization for flights occurring within a short time period over North Rhine-Westphalia. It can be used to analyze peak traffic periods, assess airport congestion, and support airspace management decisions.

temporal - range - aggregation

Query 4 Compute the number of flights passing through a specified county.

```
SELECT c.name, COUNT(*) as flight_count
FROM counties c
JOIN flights f
ON f.traj && c.geom
WHERE c.name = :county AND st_intersects(f.traj, c.geom)
GROUP BY c.name;
```

In this case, the overlaps operator (`&&`) is applied to the bounding box of the spatial dimension of the `trip` column, as `c.geom` represents the spatial extent of the specified county, i.e., the county's polygon. This query can provide insights into which counties experience the most air traffic and may therefore be more affected by environmental pollution.

spatial - range - aggregation

Query 5 Retrieve flight points in a specified radius of large cities below a specified altitude.

```
SELECT p.flightid, p.altitude, p.airplanetype, p.timestamp
FROM flightpoints p
JOIN cities c
ON ST_DWithin(p.geom, c.geom, :radius)
WHERE p.altitude <= :low_altitude
AND c.population >= 200000;
```

This query operates on the `flightpoints` table, which stores captured, non-interpolated flight data points as individual entries before they are aggregated into flight tracks. Unlike previous queries that analyze entire flight tracks, this query retrieves individual data points. This allows us to assess how MobilityDB performs when handling single data points compared to queries that are based on *temporal types*. The query provides insights into the extent of noise pollution around major cities.

spatial - point - no aggregation

Query 6 Retrieve flights within a specified distance from a specified point, ordered by proximity.

```
SELECT f.flightid, f.airplaneType, f.origin, f.destination,
       ST_Distance(f.traj, ST_GeogFromText('SRID=4326;:point')) AS min_dist
FROM flights f
WHERE ST_DWithin(f.traj, ST_GeogFromText('SRID=4326;:point'), :distance)
ORDER BY min_dist ASC;
```

This spatial query does not rely on a point or polygon from one of the regional tables but instead constructs a point from a longitude-latitude pair within the NRW area, which is injected into the query by the *Benchmarking Client*. This query can be used, for example, to identify flights that have approached sensitive locations or entered no-fly zones.

spatial - range - no aggregation

Query 7 Retrieve flight tracks that intersect a given county during a specified period.

```
SELECT f.flightid, f.track
FROM flights f, counties c
WHERE c.name = :county
AND f.trip && stbox(c.geom, :period_medium)
AND eintersects(attime(f.trip, :period_medium), c.geom);
```

In this spatiotemporal query, the overlap operator (`&&`) applies both spatial and temporal filtering to the `trip` column, as `stbox()` constructs a spatiotemporal bounding box that combines the temporal extent of the specified period with the spatial extent of the county's polygon. This coarse filtering step reduces the number of candidate flights before applying the more computationally expensive `eintersects` function on `f.trip`. The bounding box predicates are meant for invoking indexes in the query plans, thus increasing query performance [3]. The query can be used to identify periods of high flight traffic in specific counties. Figure 3.7 illustrates exemplarily flight tracks retrieved by this query.

spatiotemporal - range - no aggregation

Query 8 Count the number of flights intersecting each district at a specific instant.

```
SELECT b.name, COUNT(*)
FROM flights f, districts d
WHERE f.trip && stbox(d.geom, :instant)
AND eintersects(attime(f.trip, :instant), d.geom)
GROUP BY b.name;
```

The query retrieves the number of flights occurring at a specific instant for each of the five districts within North Rhine-Westphalia. Thus, it can be used to support airspace management or monitor flight density.

spatiotemporal - point - aggregation

Query 9 Retrieve flights that passed within a given radius of a specified city during a time period.

```
SELECT f.flightId, f.track, f.airplanetype, f.origin, f.destination
FROM cities AS c
JOIN flights AS f
ON eDwithin(attime(f.trip, :period_medium), c.geom, :radius)
WHERE c.name = :city
AND f.trip && :period_medium;
```

MobilityDB's `eDWithin()` function determines whether a moving object has ever entered a specified radius around a point. This query can be used, for example, to analyze air traffic in urban planning, particularly concerning noise pollution.

spatiotemporal - range - no aggregation

Query 10 Compute the total time a flight remained below a given altitude while passing over a specified municipality during a time period.

```
SELECT m.name, f.flightid, f.track, f.origin, f.destination, f.airplanetype,
       duration(whenTrue(attime(f.altitude, :period_medium) #< :low_altitude
       & tintersects(attime(tgeompoint(f.trip), :period_medium),
       geometry(m.geom)))) AS totalTimeBelowAltitude
FROM municipalities m
JOIN flights f
ON f.trip && stbox(m.geom, :period_medium)
WHERE m.name = :municipality
AND duration(whenTrue(attime(f.altitude, :period_medium) #< :low_altitude
       & tintersects(attime(tgeompoint(f.trip), :period_medium),
       geometry(m.geom)))) IS NOT NULL;
```

In this query, the temporal less than operator (`#>`) returns a `tbool`, indicating when a flight's altitude—the *base type* of the `altitude` column—falls below a specified threshold. The `tintersects()` function produces a `tbool` representing when the flight was within the airspace of a given municipality. Both functions operate only on the portion of flights occurring within the specified period. The temporal AND operator (`&`) then combines these conditions, yielding a `tbool` that identifies time intervals when the flight was both below the altitude threshold and within the municipality's airspace. The `whenTrue()` function extracts these intervals, while the `duration()` function sums their total time. We must cast `f.trip` (of type `tgeogpoint`) to `tgeompoint` and `c.geom` (of type `geography`) to `geometry`, as the function `tintersects()` currently does not support `geography` types. This query enables environmental impact analysis, helping assess noise and pollution from air traffic over municipalities in North Rhine-Westphalia.

spatiotemporal - range - no aggregation

Query 11 Count the number of active flights over a specified municipality for each hour of a given day.

```
WITH hours AS (
    SELECT generate_series(
        DATE :day::timestamp,
        DATE :day + INTERVAL '1 day' - INTERVAL '1 second',
        INTERVAL '1 hour') AS start_time)
SELECT h.start_time, COUNT(f.flightid) AS active_flights
FROM hours h
JOIN municipalities m ON m.name = :municipality
LEFT JOIN flights f
ON eintersects(attime(f.trip, span(h.start_time, h.start_time +
INTERVAL '1 hour' - INTERVAL '1 second')), m.geom)
WHERE f.trip && stbox(m.geom, span(h.start_time, h.start_time +
INTERVAL '1 hour' - INTERVAL '1 second'))
GROUP BY h.start_time
ORDER BY h.start_time;
```

The common table expression (CTE) constructs hourly intervals for a specified day, which are then used to analyze flight traffic within these hours for a specified municipality. As the query analyses the active hourly flights of a specified day for a municipality, it can be used for flight pattern monitoring.

spatiotemporal - range - aggregation

Query 12 Retrieve flights associated with with landings or departures in cities in North Rhine-Westphalia, passing through a specified county within a specified period.

```
WITH relevant_airports AS (
    SELECT a.icao as icao
    FROM airports a
    JOIN cities c ON a.city = c.name
)
SELECT f.flightid, f.airplanetype, f.origin AS origin_airport,
       a1.City AS origin_city, f.destination AS destination_airport,
       a2.City AS destination_city
FROM flights f
JOIN counties c ON c.name = :county
LEFT JOIN airports a1 ON f.origin = a1.icao
LEFT JOIN airports a2 ON f.destination = a2.icao
WHERE (f.origin IN (SELECT icao FROM relevant_airports)
       OR f.destination IN (SELECT icao FROM relevant_airports))
AND f.trip && stbox(c.geom, :period_short)
AND eintersects(attime(f.trip, :period_short), c.geom);
```

Here, the CTE utilizes the hashed indexes that we build on the `icao` column of the `airports` table and the name column of the `cities` table. This query can provide information about which flights in the airspace of a county during a specified period actually

depart or arrive at a local airport. It can be used to assess airport connectivity, analyze regional flight patterns, or support local air traffic management.

spatiotemporal - range - no aggregation

Figure 3.7: Exemplarily flight tracks retrieved by **Query 7** for the county "Soest"

3.6 Benchmark Execution

We evaluate the performance of both SUTs in two configurations: a single-node setup and a distributed cluster setup comprising three nodes. All database configurations—including the single-node setups of MongoDB and MobilityDB, as well as their respective cluster deployments—are hosted on Google Compute Engine (GCE) instances, i.e., virtual machines (VMs), within the Google Cloud Platform (GCP).

For all setups, we use the *c3-standard-8* instance type, which provides 32GB of RAM and 8 vCPUs. Additionally, each machine is equipped with SSD storage: 500GB for single-node setups and 200GB for distributed clusters. The machines are deployed within a Virtual Private Cloud (VPC) network providing low latency and high bandwidth between the instances. Each VM runs Ubuntu Pro 22.04 LTS as its operating system. We use MongoDB version 8.0 and MobilityDB version 1.3.

In single-node setups, we deploy MongoDB as a standalone process and MobilityDB without the Citus extension. In the distributed setup of MobilityDB, we deploy the *coordinator* node on one machine, while the other two machines host the *worker* nodes. Additionally, we configure PostgreSQL according to the recommended settings based on the available VM resources. For MongoDB, we deploy *shard server*, *config server*, and *router* as independently running processes on each VM. While MobilityDB's *Benchmarking Client* connects only to the *coordinator node* to send requests, MongoDB's *Benchmarking Client* connects to all deployed *routers*, thereby reaching all nodes of the cluster.

Ideally, the *Benchmarking Client* should be distributed to better simulate a realistic application scenario, where queries typically originate from multiple machines rather than

a single one. Furthermore, since we also benchmark distributed database setups, potentially spanning multiple geographic regions, their clients should likewise be distributed [10]. However, setting up the databases in single-node and cluster setups and conducting multiple benchmark runs for each setup is costly and puts a strain on our limited financial budget. Therefore, we deploy the *Benchmarking Client* on a single VM of type *c3-standard-8*. To simulate multiple concurrent client connections to the databases, we utilize multiple threads on this client, each establishing connections and sending requests to the SUTs in parallel. Nevertheless, our implementation supports deploying the *Benchmarking Client* across multiple machines. The *Benchmarking Manager* can be adapted to interact with multiple *Benchmarking Client* instances and configure them individually by modifying their respective YAML files.

Before running the benchmarks, we conduct a preparation phase. During this phase, we load flight data points, municipality data, county data, district data, and airport data into tables or collections. We convert longitude-latitude pairs into `geography` types for MobilityDB and GeoJSON types for MongoDB. Subsequently, we interpolate the flight data points and create the trajectories for the flight tracks. For MongoDB, we insert the interpolated flight data points into the time-series collection, while for MobilityDB, we aggregate individual flight data points into flight tracks and insert them into the `flights` table. For the distributed setups, we distribute tables and collections as described in sections 3.3 and 3.4. The tables and collections containing regional and airport data are replicated across all nodes in both SUTs, ensuring that each node maintains a complete copy. Since these tables and collections contain relatively small amounts of data, the additional storage requirements per node are minimal. Replicating this data across all nodes eliminates the need for data re-distribution during JOIN operations, as these operations can be performed on co-located tables and collections. For this reason, Citus also by default enforces table co-location for JOIN operations [22]. Finally, we create indexes to optimize query performance.

We benchmark the two different setups of both SUTs using two approaches. First, for our application-centric benchmark, we shuffle the queries to simulate realistic query behavior. Second, to evaluate the SUTs' performance on different query types and individual queries, we conduct category benchmarks, executing spatial and spatiotemporal queries separately and running queries of the same type sequentially.

For both single-node and cluster setups, benchmark runs begin with a series of warm-up requests before the benchmark execution starts. This is done to stabilize the quality behavior of the SUTs, as newly spawned systems may exhibit different performance characteristics compared to systems that have been running for some time. For comparability reasons, we focus on the stabilized state. The differing performance of databases can, e.g., be due to caches that have to be filled [10].

Following the warm-up phase, we proceed with the actual experiments. During benchmark execution, we closely monitor the resource utilization of the *Benchmarking Client* to ensure the client does not become a bottleneck. When executing our benchmarks, we aim to fully utilize the SUT. To achieve this we adjust the number of threads of the *Benchmarking Client*, regulating the generated load to effectively stress the SUT and derive meaningful performance insights [10].

For single-node setups of MongoDB and MobilityDB we use 16 threads on the *Benchmarking Client*. In the application-centric benchmark, these threads collectively execute each of the 12 queries 20 times, totaling 240 requests. The client ingests changing pa-

rameter values for the queries in a pseudo-random manner. Using these configuration both systems' CPUs are fully utilized during the entire benchmark execution. In the category benchmarks, we execute each spatial query sequentially 100 times, totaling 300 queries. Similarly, to evaluate the SUTs for a spatiotemporal workload, we execute each spatiotemporal query sequentially 100 times, resulting in a total of 600 queries.

For the distributed setup of MongoDB, we increase the number of client threads to 32, as it can handle a significantly higher load compared to the single-node setup. To maintain a meaningful benchmark execution time despite the increased parallelism, we extend the benchmark runtime by increasing the number of executed queries. In the application-centric benchmark, we execute each of the 12 queries 150 times, resulting in a total of 1,800 requests sent to the database. Similarly, in the category benchmarks, we execute each query 300 times, leading to 900 queries in the spatial category benchmark and 1,800 queries in the spatiotemporal category benchmark.

For the distributed setup of MobilityDB, we reduce the number of client threads to 8 while still generating sufficient load on the SUT. The CPUs of the *worker* nodes in the Citus cluster remain fully utilized throughout the benchmark execution. For consistency, we use the same number of queries in both the application-centric and category benchmarks as in the MongoDB cluster benchmarks.

We repeat each benchmark run, whether from the application-centric, spatial category, or spatiotemporal category benchmark, three times for both SUTs and for both single-node and cluster setups, resulting in a total of 36 runs. This ensures the repeatability and reproducibility of our benchmarks [10].

4 Evaluation

Next, we evaluate the benchmark results for both single-node and cluster setups of MongoDB and MobilityDB. We executed each benchmark configuration three times to ensure repeatability and reproducibility. For any given configuration, we observed only small variations in total execution time, never exceeding 5%. This variability may stem from several factors, including inherent performance fluctuations in cloud environments [32]. Since all runs yielded similar results, we present only the first execution of each benchmark. The results of the benchmarks runs can be found on GitHub[11].

4.1 Single-node setups

To execute the queries, on the single-node setups of MongoDB and MobilityDB, we started with a small number of threads on the *Benchmarking Client* and gradually increased them until the database servers reached full CPU utilization. Using 16 threads on the *Benchmarking Client* ensured a sufficient number of concurrent connections sending queries to fully utilize the SUTs. In the application-centric benchmark runs, we executed a total of 240 queries against the databases, with each of the 12 queries running 20 times. In our application-centric benchmark runs, queries are intentionally executed in an unordered manner to better simulate a realistic workload.

Table 4.3 presents the performance of both SUTs in single-node setups under an application-centric workload generated by 16 client threads. We report the total execution times in seconds alongside the corresponding query throughput (queries executed per second). Additionally, we provide latencies in seconds for the 90th ($\eta_{.90}$) and 99th ($\eta_{.99}$) percentiles, as well as the average latency, all computed across the full set of 240 queries. The results show that MobilityDB's 90th percentile execution time is approximately 55% higher than MongoDB's, while its 99th percentile execution time exceeds MongoDB's by about 48%. Moreover, MobilityDB's average latency is 39% greater than that of MongoDB. Overall, MongoDB outperforms MobilityDB across all key metrics, achieving lower execution times, reduced latencies, and higher query throughput.

Table 4.3: Performance comparison of single-node setups of MongoDB and MobilityDB

Database	**Execution time (s)**	$\eta_{.90}$ **(s)**	$\eta_{.99}$ **(s)**	**Avg. latency (s)**
MongoDB	775 (0.31 q/s)	167	332	44
MobilityDB	1024 (0.23 q/s)	258	490	61

MongoDB was able to provide a shorter execution time to run all 240 queries than MobilityDB, executing them in approximately 75.7% of the time required by MobilityDB.

However, we observe that different query types exhibit varying performance across both SUTs, as shown in Table 4.4. Notably, MongoDB outperforms MobilityDB in both temporal and spatial queries. In the application-centric benchmark on single-node setups, MobilityDB's average latency for temporal queries is approximately 2.1 times higher than MongoDB's (2.7 s vs. 1.3 s), while for spatial queries, the difference is even more pronounced, with MobilityDB showing an average latency 3.6 times higher (128.0 s vs. 35.6 s).

[11] `https://github.com/FelixMedicusf/master_project`

Table 4.4: Performance comparisons of single-node setups of MongoDB and MobilityDB for different query types

Database	Query Type	$\eta_{.90}$ (s)	$\eta_{.99}$ (s)	Avg. latency (s)
MongoDB	Temporal	3.4	5.9	1.3
	Spatial	144.5	206.5	35.6
	Spatiotemporal	210.0	345.7	70.1
MobilityDB	Temporal	7.9	14.1	2.7
	Spatial	379.1	517.6	128.0
	Spatiotemporal	152.3	492.3	57.2

Latencies vary substantially across query types, with the differences between the two SUTs being particularly pronounced for spatial queries.

For spatiotemporal queries, however, MobilityDB performed slightly better, with a lower average latency of 57.2 s, compared to 70.1 s for MongoDB.

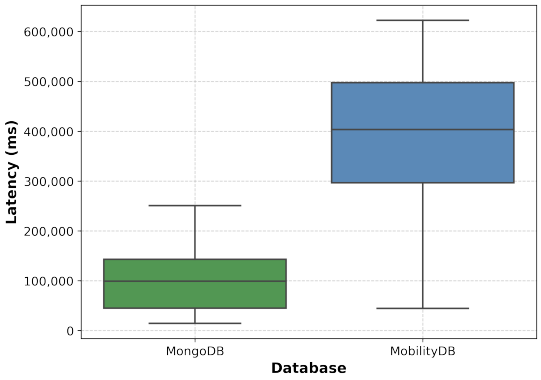
Therefore, to evaluate the performance of both SUTs across different query types, we conducted category benchmarks for both SUTs, executing spatial and spatiotemporal queries separately against the databases. Moreover, we executed the queries sequentially to observe how the databases handle both different query types and specific queries within each type.

Executing each of the three spatial queries 100 times, resulting in a total of 300 queries, required 610 seconds for MongoDB and 2,724 seconds for MobilityDB. Consequently, MongoDB's throughput was more than four times higher than that of MobilityDB. MongoDB's superior performance on spatial queries is also reflected in the boxplots in Figure 4.8, 4.9, and 4.10, which present the latency distributions for spatial queries 4, 5, and 6, respectively. To evaluate database performance on specific queries of spatial and spatiotemporal type, we use the middle 80% of the respective queries—in this case, the central 80 queries. We intentionally exclude the first and last 10% of queries from our latency analysis, as their execution times may be influenced by different preceding or subsequent queries. Furthermore, we also exclude latency measurements from the analysis that are identified as outliers, as determined using the boxplot method, which detects values that fall outside 1.5 times the interquartile range (IQR).

The boxplots for spatial queries clearly indicate that MongoDB outperforms MobilityDB across all spatial queries. The median latencies for Query 4 and Query 5 are approximately 4 to 5 times higher in MobilityDB. The variance in latencies for Query 4 in both databases may be influenced by the county used in the query. Some counties, particularly those containing major airports, experience high air traffic, while others, which are not located along popular flight routes, show significantly lower air traffic. This variation in the number of flight tracks across different counties likely contributes to the observed latency variance. Query 5 exhibits significantly lower latencies across both SUTs compared to the other two spatial queries. We assume this is because Query 5 operates on the `flightpoints` collection in MongoDB and the `flightpoints` table in MobilityDB, both of which store non-interpolated flight data points, resulting in a significantly smaller data volume. Furthermore, each flight data point in MobilityDB is stored in a single row and is

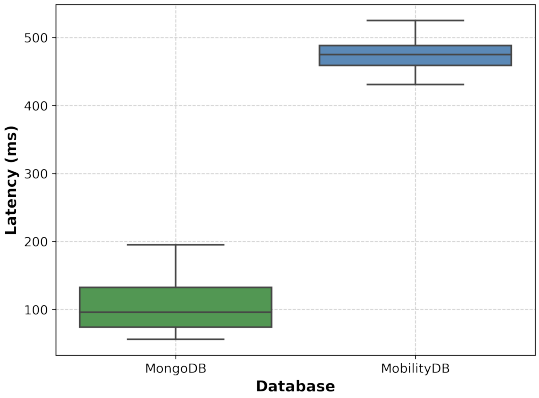
not aggregated into temporal types representing flight trips, as is the case for the `flights` table. The performance difference for spatial queries becomes especially evident in Query 6, where the median latency for MobilityDB is approximately 15 times higher than that of MongoDB.

Figure 4.8: Spatial Query 4 latency distributions



The median latency for single-node MobilityDB is approximately four times higher than that of single-node MongoDB.

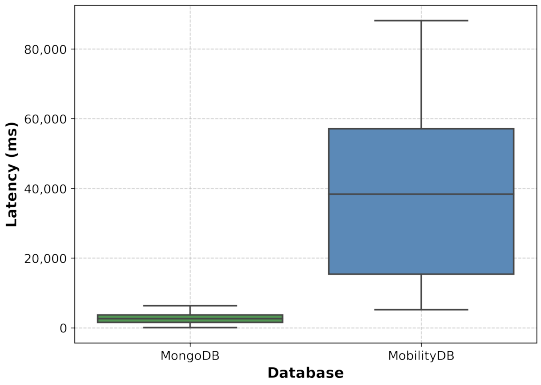
Figure 4.9: Spatial Query 5 latency distributions



MongoDB outperforms MobilityDB, with latencies for all executions of Query 4 being lower than those of MobilityDB.

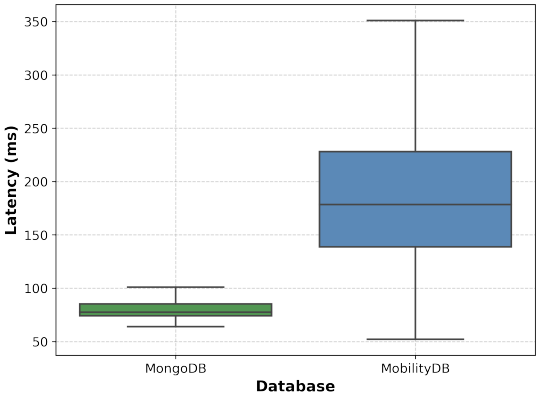
Executing the 600 spatiotemporal queries took in total 2581 seconds for MongoDB and 2047 seconds for MobilityDB. In case of the spatiotemporal queries, MobilityDB's throughput was approximately 1.26 higher than the throughput of MongoDB. However, the average latency advantage for the different queries varied across both SUTs, as shown by Figure 4.12. Figure 4.12 shows the average latencies for the spatiotemporal queries except for Query 8, which exhibits substantially lower latencies across both SUTs. This is due to the fact that, unlike the other spatiotemporal queries, Query 8's temporal selectivity is constrained to a single point in time (timestamp) rather than a time period. The latency distribution for Query 8 is presented separately as a boxplot in Figure 4.11.

Figure 4.10: Spatial Query 6 latency distributions



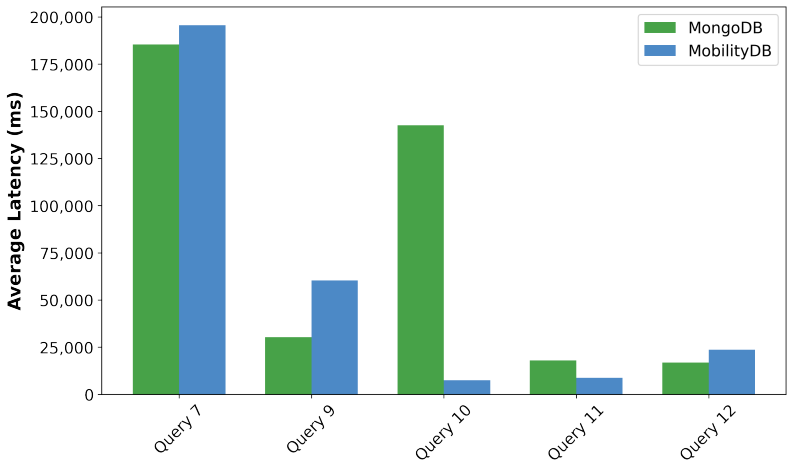
For Query 6 MongoDB exhibits significantly lower latencies and variance.

Figure 4.11: Spatiotemporal Query 8 latency distributions



Query 8 shows reduced latencies for both SUTs compared to other spatiotemporal queries.

Figure 4.12: Average latencies for spatiotemporal queries



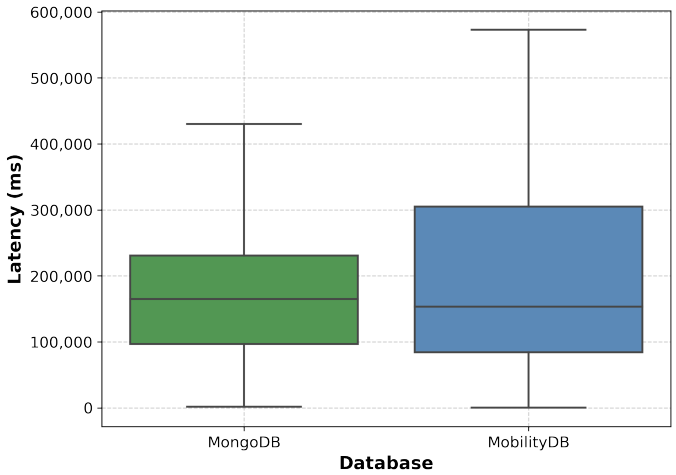
For spatiotemporal queries, whether MongoDB or MobilityDB has the lower average latency depends on the specific query.

Looking at Figure 4.12, Query 10 stands out, as it exhibits significantly higher latencies for MongoDB compared to MobilityDB. Recall that Query 10 retrieves the duration airplanes spend flying at low altitude within a municipality during a period. We attribute MongoDB's higher latencies for this query to computational overhead—while MobilityDB uses the built-in `duration()` function, MongoDB must perform computationally expensive `$reduce` operations on aggregated arrays of flight data points.

Query 9, which retrieves flight tracks occurring within a certain radius around a city during a period, has approximately twice the average latency on MobilityDB compared to MongoDB. In contrast, the average latency of Query 11, which counts the number of active flights per hour located within a municipality, is about two times higher for MongoDB.

Notably, Query 7 exhibits substantially higher average latencies for both SUTs compared to other spatiotemporal queries. A possible explanation is that Query 7 employs spatial filtering based on counties, which generally cover larger geographic areas than municipalities, potentially requiring the evaluation of more flight tracks. Furthermore, Query 7 applies a random temporal constraint spanning 2 to 15 days (`period_medium`), whereas Query 12, which also filters by counties, covers a shorter random period of 0 to 2 days (`period_short`). We take a closer look on the latency distributions of Query 7 in Figure 4.13. The median latency of MobilityDB seems to be slightly lower than MongoDB's median latency, however, MobilityDB shows a larger variance for the latencies. The maximum latency observed for MobilityDB (approximately 573 s) occurred for a query covering a period of roughly 11 days (`[2023-09-19 06:13:07, 2023-09-30 07:00:56]`), while MongoDB's maximum latency (approximately 430 s) resulted from a query spanning around 15 days (`[2023-04-14 19:15:33, 2023-04-29 13:24:35]`). Conversely, we recorded the minimum latencies for both SUTs using identical parameter pairs, each covering a much shorter interval of approximately 21 hours. This observation aligns with our expectations, as shorter temporal periods filter out more flight tracks before the spatial constraint is applied, reducing the number of tracks that need to be checked for whether they fall within the county.

Figure 4.13: Spatiotemporal Query 7 latency distributions



Query 7 exhibits similar median latencies for both SUTs, while MobilityDB shows greater latency variance.

Overall, considering all query types, MongoDB generally outperforms MobilityDB in the single-node setup. However, this performance advantage varies depending on the query type. While MongoDB demonstrates better performance for purely temporal and spatial queries, the results for spatiotemporal queries are mixed, with MobilityDB performing better for some queries and MongoDB for others. Notably, MobilityDB exhibits a particular better performance on a spatiotemporal query for which MongoDB required a workaround implementation, whereas MobilityDB probably could address this query more efficiently through a built-in temporal function.

4.2 Cluster setups

For the cluster setups, we aimed to evaluate how the systems handle an increased load. Therefore, we tried to raise the number of threads on the *Benchmarking Clients*, effectively increasing the number of concurrent connections to the databases. For MongoDB, we increased the thread count from 16 to 32, achieving high CPU utilization across all database nodes throughout the benchmark execution. However, the nodes' CPU utilization varied during benchmark execution since, in our distributed MongoDB setup, temporal and spatiotemporal queries can be routed by MongoDB's *routers* directly to *shard servers* holding the relevant data based on the query's temporal constraint.

For MobilityDB, we had to reduce the number of threads on the *Benchmarking Client*. Using 16 or more threads, certain queries experienced prolonged execution times, causing the database to become unresponsive to the client and leaving the connection idle. To prevent this issue, we had to reduce the client's query count to 8. However, even with just 8 threads, we generate a high load on the *worker* nodes of the MobilityDB cluster, ensuring that the SUT is properly stressed.

Figure 4.5 presents the thread count, benchmark execution times, throughput, and latency metrics for both SUTs in distributed setups comprising three nodes. As shown, executing the 1,800 queries in the first benchmarking approach is substantially faster in MongoDB, as it can handle a higher number of concurrent connections. Consequently, MongoDB also achieves a substantially higher throughput of approximately 1.66 queries per second.

While latencies are slightly lower for MobilityDB, this should be interpreted with caution, as the reduction in latency is likely a consequence of its lower throughput and number of concurrent connections, given the inverse relationship between latency and throughput [10].

Table 4.5: Performance comparison of distributed setups of MongoDB and MobilityDB

Database	# Threads	Execution time (s)	$\eta_{.90}$ (s)	$\eta_{.99}$ (s)	Avg. lat. (s)
MongoDB	32	1087 (1.66 q/s)	58	172	18
MobilityDB	8	3463 (0.52 q/s)	62	131	15

MongoDB was able to provide a shorter execution time to run all 1800 queries than MobilityDB, executing them in approximately 31.4% of the time required by MobilityDB.

Comparing Table 4.5 to Table 4.3 it becomes clear that both SUTs, which were fully loaded in both setups, benefit from adding database nodes, i.e., horizontal scaling. MongoDB benefits considerably, as its throughput increases more than fivefold, from 0.31 queries per second to 1.61 queries per second. Additionally, latencies decrease substantially. The 90th percentile latency for all query types drops from 167 seconds to just 58 seconds, while the 99th percentile latency decreases from 332 seconds to 172 seconds. Moreover, the average latency in the cluster setup is reduced by more than half, from 41 seconds to 18 seconds.

For the single-node setup, we established 16 concurrent connections using 16 threads on the *Benchmarking Client*. However, for the distributed setup, we established only 8 concurrent connections to the database. Nevertheless, the throughput of distributed MongoDB is more than twice that of the single-node MobilityDB (0.52 queries per second vs. 0.23 queries per second). We assume this is because queries in the distributed setup of MobilityDB exhibit a higher frequency of latency outliers compared to the single-node setup of MongoDB, as also the overall benchmark execution time is significantly longer. As a result, when a query takes too long to terminate, the connection from a thread to the database becomes idle instead of actively responding, preventing the thread from sending a new query. However, despite the higher throughput, latencies significantly decreased in the distributed setup of MobilityDB, indicating that MobilityDB also benefited from horizontal scaling. The 90th percentile latency decreased from 258 to 62 seconds, the 99th percentile latency from 490 to 131 seconds, and the average latency dropped significantly from 61 to 15 seconds.

Similarly to single-node setups, in the distributed setups of the SUTs, the databases' performance heavily depends on query types, as shown by the great variations in latencies across different query types in Table 4.6. Once again, it is important to note that comparing latencies between MongoDB and MobilityDB in distributed setups does not provide a representative measure of performance, as MongoDB handles four times the number of concurrent connections, resulting in a significantly higher load.

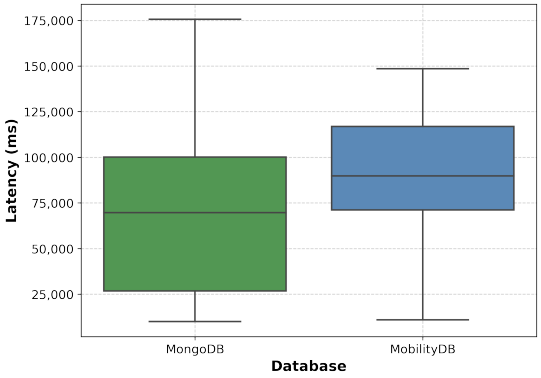
Table 4.6: Performance comparisons of distributed MongoDB and MobilityDB for different query types

Database	Query Type	$\eta_{.90}$ (s)	$\eta_{.99}$ (s)	Avg. latency (s)
MongoDB	Temporal	2.0	3.9	0.8
	Spatial	46.7	83.0	12.0
	Spatiotemporal	93.0	187.4	30.5
MobilityDB	Temporal	1.4	2.2	0.7
	Spatial	99.6	137.2	31.3
	Spatiotemporal	35.3	129.4	14.6

Latencies vary substantially across query types for both SUTs. However, a direct latency comparison between SUTs in distributed setups can hardly serve as a performance reference, as MongoDB processed a significantly higher concurrent query load.

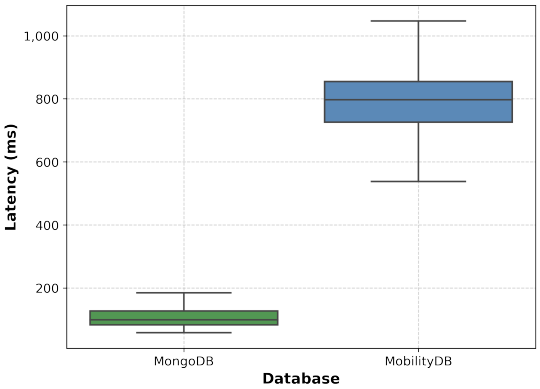
Comparing Table 4.6 with Table 4.4, we observe that all latency metrics for both SUTs are lower in the distributed setups. Despite the higher throughput, the average latencies for temporal, spatial, and spatiotemporal queries in the distributed MobilityDB setup are only approximately one-fourth of those in the single-node setup. For MongoDB, the latencies of spatial queries decrease substantially, dropping to approximately one-third of their values in the single-node setup. The great decrease in average latencies for spatial queries in MongoDB may be attributed to the fact that, while temporal and spatiotemporal queries are typically routed to and processed on a single shard server, spatial queries are executed in parallel across all *shard servers*, with each processing a subset of the respective collection before the results are merged.

Figure 4.14: Spatial Query 4 latency distributions



Distributed MobilityDB exhibits a higher median latency, despite handling fewer concurrent connections and queries.

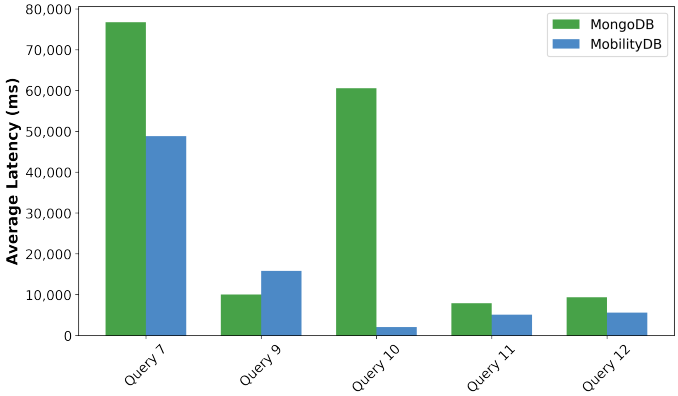
Figure 4.15: Spatial Query 5 latency distributions



For Query 5, the median latency of distributed MobilityDB is approximately five times higher than that of distributed MongoDB

To evaluate the performance of distributed MobilityDB and MongoDB across different query types and specific queries, we also conducted category benchmarks for the distributed setups of both SUTs. Each spatial and spatiotemporal query in the category benchmark for the distributed setups is executed sequentially 300 times with changing parameter values. Therefore, boxplots and averages are calculated based on 240 execu-

Figure 4.16: Average latencies of spatiotemporal queries for distributed setups



MobilityDB shows lower average latency for four out of six spatiotemporal queries. However, this is likely due to the lower number of concurrent queries that MobilityDB needs to handle.

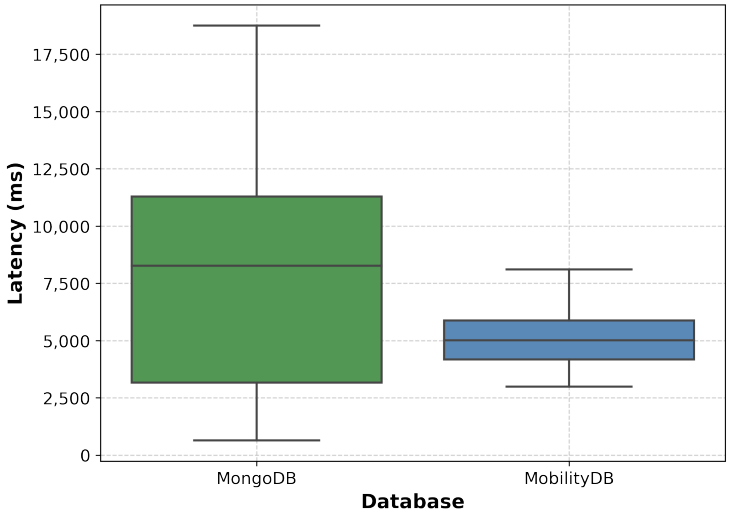
tions of each query, assuming no outliers are detected. The execution of spatial queries required 650 seconds for MongoDB, whereas MobilityDB needed nearly six times longer, specifically 3,855 seconds.

Executing 1,800 spatiotemporal queries required 1,786 seconds for MongoDB and 3,161 seconds for MobilityDB. For both query types, MongoDB achieves a substantially higher throughput compared to MobilityDB: 2.77 queries per second vs. 0.47 queries per second for spatial queries, and 1.01 queries per second vs. 0.57 queries per second for spatiotemporal queries. Figure 4.14 and 4.15 show the latency distributions for distributed setups of MongoDB and MobilityDB for the spatial queries 4 and 5, respectively. Despite MongoDB's substantially higher throughput and greater number of concurrent connections, its latencies overall remain lower for both queries. For Query 4, however, MongoDB shows a wider spread of higher latency values, suggesting greater variability and occasional latency spikes compared to MobilityDB.

Figure 4.16 shows the average latencies for the spatiotemporal queries. Once again, Query 8 is not included in the figure due to its substantially lower average latency. Its average latencies are 93 milliseconds for MongoDB and 285 milliseconds for MobilityDB. For 4 out of the 6 spatiotemporal queries, MobilityDB has a lower average latency. However, if the number of concurrent connections to distributed MongoDB were reduced to 8—matching MobilityDB—and thus generating a substantially lower load on MongoDB, its latencies across all queries would likely be significantly lower. The latency distributions for Query 11 and Query 12 are shown in Figure 4.17 and Figure 4.18, respectively. In both cases, latencies are lower for MobilityDB. Additionally, MongoDB exhibits greater variability in latency for both queries compared to MobilityDB.

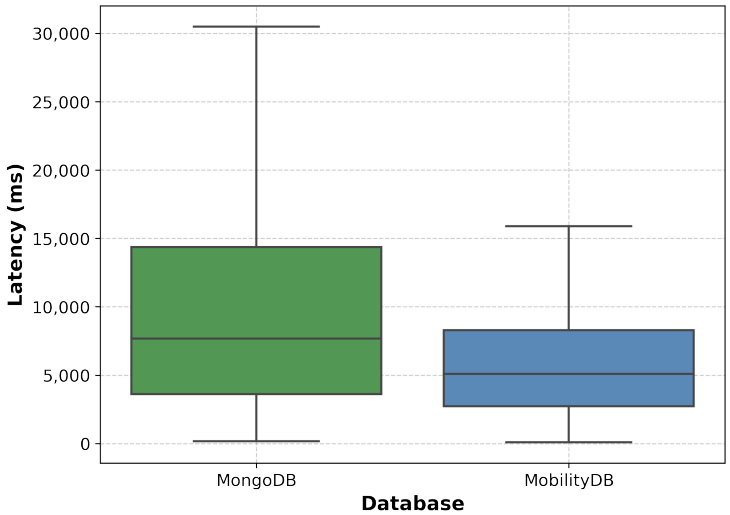
Overall, both systems benefit substantially from scaling out, as they process more queries per second and experience reduced latencies. MongoDB, in particular, saw substantial gains from the addition of two nodes, achieving more than a fivefold increase in throughput and a reduction of average latency by more than half across all query types. MobilityDB's throughput in the distributed setup increased by approximately 67.7%, while its average latency was reduced by more than 4 times, dropping from 61 seconds to 15 seconds.

Figure 4.17: Spatiotemporal Query 11 latency distributions



For Query 11, MongoDB exhibits a higher median latency and a higher latency variance than MobilityDB.

Figure 4.18: Spatiotemporal Query 12 latency distributions



For Query 12, the median latency of MongoDB is approximately 66.2% higher than the median latency of MobilityDB.

5 Discussion

Spatiotemporal databases face the challenge of providing scalable storage and spatiotemporal querying solutions for large volumes of historic mobility data [4]. However, the storage and querying capabilities for temporal, spatial, and spatiotemporal data differ significantly between our two SUTs, MobilityDB and MongoDB. Compared to MongoDB, MobilityDB natively provides a more comprehensive suite of data types, operators, and functions specifically designed for handling temporal, spatial, and spatiotemporal data. Furthermore, using MobilityDB's temporal types can dramatically reduce storage space [2].

A major limitation of spatial and spatiotemporal queries in MongoDB is that its spatial functions—`$geoWithin` for containment checks, `$geoIntersects` for intersection checks, and `$geoNear` for distance calculations—cannot take the geometry's coordinates from expressions[12]. As a result, these functions cannot directly use coordinates retrieved in earlier stages of an aggregation pipeline. Consequently, all spatial and spatiotemporal queries in our evaluation—except for Query 6—required two separate queries in MongoDB. The first query retrieves the coordinates of the relevant municipality, county, or district, while the second applies the spatial or spatiotemporal query logic using those coordinates. Inserting random polygon coordinates for spatial and spatiotemporal queries would eliminate the need for two separate queries in MongoDB. However, by using the actual geographic boundaries of regions within the evaluated area, we enhance the realism and relevance of the benchmarks. The limitations of MongoDB's geospatial functions also influences the kind of queries that we use in our benchmarks. For example, in air traffic control, incident analysis may involve detecting cases where two airplanes came into close proximity. In MobilityDB, this can be achieved efficiently with a single operator. In contrast, MongoDB lacks native spatiotemporal operators and functions, and its spatial functions are limited in their functionality, making such a query infeasible without adding custom functions. Some queries involving operations recommended by OGC for Moving Features are difficult to implement. One such example is a query using the OGC-recommended *cumulativeDistanceAtTime* operator, which, in our case, computes the cumulative distance traveled by airplanes. While MobilityDB supports this with the `cumulativeLength(tpoint)` function, MongoDB lacks an equivalent built-in feature. Given MongoDB's limited support for spatiotemporal queries, we still employ a diverse set of queries reflecting real-world scenarios. Nevertheless, implementing custom functions for MongoDB could further expand the range of these queries and cover more of the operations that are recommended by the OGC for moving features.

Two of our three spatial queries are based on the flight tracks' trajectories which are of type *LineString*. In MobilityDB, trajectories are precomputed and cached for temporal points and can be invoked during query execution using the `trajectory()` function on a linearly interpolated `tpoint`, providing greater flexibility [3]. For MongoDB, however, we had to construct the trajectories on the client side, as MongoDB does not natively support a function for building *LineStrings*.

For our benchmark, we use WGS 84 to store geographic data in latitude and longitude. However, the raw flight dataset uses the the European Terrestrial Reference System 1989 (ETRS89) to express locations, as the flight tracks are only observed in a constrained region, specifically the state of North Rhine-Westphalia. ETRS89 projects geographic

[12] `https://jira.mongodb.org/browse/SERVER-37675`

coordinates into a Cartesian system, such as UTM zone 32N. UTM zone 32N covers all parts of Germany and represents locations in meters rather than in degrees. However, since MongoDB does not support ETRS89, we opt for WGS 84 for both databases to ensure a fair comparison between both SUTs. MongoDB's `2dsphere` index only supports spatial data in geodetic coordinates , thus longitude and latitude. However, many geographic applications at the city or regional scale rely on planar Cartesian coordinates [28]. Using ETRS89 for MobilityDB could improve the performance of the database for spatial operations, as ETRS89 operates in a Cartesian coordinate system, where calculations are computationally simpler compared to spherical computations, as required by WGS 84 [28]. We struggled with the decision of whether to use ETRS89 for MobilityDB—taking advantage of its full feature set and potentially increasing performance—or to choose WGS84 to ensure better comparability between the SUTs, albeit potentially reducing MobilityDB's performance capabilities. We opted for the latter.

In single-node setups, MongoDB shows a higher throughput and lower query latencies than MobilityDB. MongoDB, on average, performs better on all temporal and spatial queries, as well as on four out of six spatiotemporal queries. In particular, MongoDB excels in spatial queries compared to MobilityDB. Query 4 has a significantly longer execution time for both SUTs compared to the closely related Query 7, as it applies spatial filtering to flight tracks across the entire dataset rather than only to those occurring within a specific time period, as is the case for Query 7. We assume, the reason for the large differences in latencies for Query 4 and Query 6 might lie in the indexes that are used in MobilityDB. Queries 4 and 6 use operators and functions from the PostGIS extension, which MobilityDB depends on. Query 4 applies the overlap operator (`&&`) to the `traj` column, while Query 6 utilizes the PostGIS `ST_DWithin()` function on the `traj` column. These operators and functions invoke the GiST index that we build on the trajectories. Spatial indexing in PostGIS, however, only stores the bounding boxes for geometries or geographies [33]. Spatial operators and functions use the bounding boxes created by the GiST index as a primary filter to quickly determine a set of geometries or geographies potentially matching the query condition [33]. However, most spatial queries also require a secondary filter, which tests a more specific spatial condition by using a spatial predicate function. For example, in Query 4, the overlaps operator (`&&`) first checks whether the bounding box of a flight track's trajectory overlaps with the bounding box of a county's geography, which is also indexed using the GiST index [34]. If an overlap is detected, a secondary filter then verifies whether the trajectory actually intersects the county. However, when the bounding boxes of flight trajectories cover large portions of the evaluated area (the state of NRW)—as flight tracks regularly do—and the county's bounding box is also relatively large, the primary filter, where the GiST index is applied, becomes less restrictive and thus ineffective at filtering out trajectories [3]. As a result, more candidates proceed to the computationally expensive secondary filter stage. However, segmenting flight tracks into shorter segments should consequently reduce the size of their trajectories' bounding boxes, covering a smaller portion of the state of NRW. This, in turn, would allow the GiST index to perform more efficient primary filtering. This assumption is supported by [6], which demonstrates that segmenting vessel trips—stored using MobilityDB's `tpoint` data type—drastically reduces latencies for a query that is closely related to Query 4. The performance implications of segmenting temporal point types can be compared to those of segmenting trajectories, as MobilityDB's GiST index constructs bounding boxes for both data types, influencing query efficiency in a similar manner. SECONDO's spatiotemporal index, for instance, stores bounding boxes for individual trajectory segments rather

than the entire trajectory [3]. Implementing a similar approach in MobilityDB could potentially enhance its performance for queries like Query 4 by enabling a more restrictive primary filter. Storing bounding boxes for segments of temporal types, could also lead to an improved performance for spatiotemporal queries. However, storing bounding boxes for multiple smaller segments of trajectories and temporal types would also increase the index storage size.

In distributed setups, MongoDB's throughput exhibits a greater relative increase compared to MobilityDB's when measured against their respective single-node setups. This disparity could be attributed to several factors, one of which is the employed sharding strategy and its impact on query distribution. As mentioned earlier, sharding flight points based on their timestamps enables MongoDB's *routers* to efficiently distribute queries having a temporal constraints to the specific shard servers that store the relevant flight points. This efficient allocation reduces the workload on database nodes by ensuring they only process queries for data they store, thereby optimizing resource utilization. However, in MobilityDB, we shard the `flights` table over the *worker* nodes based on the value of the `flightId` column, as Citus cannot extract the temporal dimension from the `trip` column for sharding. Since temporal, spatial, and spatiotemporal constraints are defined based on the `trip` column, which is not the sharding key, the Citus *coordinator* broadcasts all queries to the *worker* nodes [22]. The query distribution behavior of MobilityDB and MongoDB can be observed in the execution plan excerpts for Query 7, shown in Figures 5.19 and 5.20, respectively. This exemplary query retrieves all flight tracks that were within the county `Krs Duisburg` during the time period `[2023-02-01 10:00:00, 2023-02-10 10:00:00]`.

In MobilityDB, the Citus *coordinator* node generates the distributed query plan, creating 16 tasks—one for each shard. The two *worker* nodes, each holding 8 shards, execute 8 tasks in parallel. Once completed, they send their results to the *coordinator* node. As seen in line 11, the *worker* nodes perform a sequential scan on the `counties` table, which is reasonable given its small size of 53 entries. Furthermore, as shown in line 23, the GiST index on the `trip` column serves as a primary filter, excluding rows where the spatiotemporal bounding box of the `trip` value does not overlap with the bounding box defined by the time period and the county's polygon.

In MongoDB, however, the same query is directed exclusively to the "shard1ReplSet" shard by the *router* deployed on the `mongodb-node-1`, as indicated by the lines 3, 8 , and 9. This shard, hosted on a specific node within the cluster, stores all documents with timestamps from the first four months of 2023. Furthermore, as seen on line 30, the $geoWithin stage of the aggregation pipeline is populated with the polygon coordinates of "Krs Duisburg", which were retrieved in a preceding query.

Another factor contributing to the greater increase in throughput for distributed MongoDB compared to distributed MobilityDB, when measured against their respective single-node setups, is the role of the clusters' nodes. In MongoDB, all three machines in the cluster store data from the sharded collections and actively participate in query processing. In contrast, MobilityDB's cluster consists of only two *worker* nodes that store data of sharded tables and handle query execution, while the third machine serves as the *coordinator* node. The *coordinator* is responsible for communicating with the client, distributing query plans, and aggregating results from the worker nodes, but it does not process queries that are issued on the sharded tables [22]. This difference is also reflected in the CPU utilization of the coordinator node, which remains low during our benchmark execution runs. As a

Figure 5.19: Distributed Query Execution Plan of MobilityDB for Query 7

```
Distributed Citus Plan (Citus Adaptive Query):
 -> Task Count: 16
 -> Tuple data received from nodes: 15 kB
 -> Tasks Shown: All

 -> Task
    -> Tuple data received from node: 928 bytes
    -> Node: host=10.132.0.3 port=5432 dbname=aviation_data

    -> Nested Loop
      -> Seq Scan on counties_102011 k
           Filter: ((name)::text = 'Krs Duisburg'::text)
           Rows Removed by Filter: 52

      -> Bitmap Heap Scan on flights_102066 f
            Recheck Cond: (trip && stbox(k.geom,
            '[2023-02-01 10:00:00+00, 2023-02-10 10:00:00+00]'::tstzspan))
            Filter: eintersects(attime(trip,
            '[2023-02-01 10:00:00+00, 2023-02-10 10:00:00+00]'::tstzspan), k.geom)
              Rows Removed by Filter: 316
              Heap Blocks: exact=351

      -> Bitmap Index Scan on idx_flights_trip_gist_102066
              Index Cond: (trip && stbox(k.geom,
              '[2023-02-01 10:00:00+00, 2023-02-10 10:00:00+00]'::tstzspan))

    Planning Time: 2.396 ms
    Execution Time: 4631.751 ms

 -> Task
   -> Tuple data received from node: 1008 bytes
   -> Node: host=10.132.0.2 port=5432 dbname=aviation_data
    ...

 ...
Planning Time: 1.607 ms
Execution Time: 4665.379 ms
```

result, adding two additional nodes to MongoDB provides more resources for direct query processing compared to MobilityDB, where one of the three nodes primarily handles coordination rather than execution. In the distributed setup of MobilityDB, we could assign a GCE instance type with significantly fewer resources to the coordinator node, given its lower requirements for CPU and RAM. For simplicity, however, we opt to use the same machine type for *coordinator* and *worker* nodes.

Since data partitioning is crucial for efficient query processing, the performance of distributed MobilityDB could potentially be improved by refining its sharding strategy. In our benchmarking approach, the chosen sharding strategy ensures temporal data locality for MongoDB [4]. However, for MobilityDB, it does not guarantee spatial, temporal, or spatiotemporal data locality for flight tracks, which may negatively impact query performance. Hierarchical partitioning allows for spatiotemporal proximity of trip data by first partitioning by the temporal extent and then by the spatial extent or vice versa. When choosing the first option, first the dataset's time extent is split into intervals of the same length. The flight tracks would then be assigned to the partitions that overlap their time extent. Afterward, inside each temporal partition a quad tree is utilized to partition the trips of flight tracks spatially. The parts in every temporal partition are then z-ordered [23]. Ultimately, a partition is created for each *worker* node. Pre-partitioning tables before distributing them across the *worker* nodes enables the Citus distributed query planner to

Figure 5.20: Distributed Query Execution Plan of MongoDB for Query 7

```
{
  "serverInfo": {
    "host": "mongodb-node-1",
    ...
  },
  "splitPipeline": null,
  "shards": {
    "shard1ReplSet": {
      "host": "10.132.0.4:27018",
      "stages": [ ... ]
        }
    },
  "command": {
    "aggregate": "system.buckets.flightpoints_ts",
    "pipeline": [
        ...
      , {"$match": {
            "timestamp": {
                "$gte": new ISODate("2023-02-01T10:00:00.000Z"),
                "$lte": new ISODate("2023-02-10T10:00:00.000Z")
                }
            }
        },
        {
        "$match": {
            "location": {
                "$geoWithin":{
                    "$geometry": {
                    "type": "Polygon",
                    "coordinates": [
                    [[
                      6.692170160029587,
                      51.55906071305452
                    ],
                    ...
                    ]]
                    }
                }
        ...
      ...
```

exclude irrelevant partitions—and consequently, unnecessary *worker* nodes—from execution plans. As a result, the *coordinator* does not always need to broadcast queries to all *worker* nodes, as in our case, but can instead route them directly to the *worker* nodes that store the relevant partitions. Hierarchical partitioning might work well for our use case, as our flight data has a temporal density pattern, with more flights occurring at the day than during the night [23].

An alternative approach to achieving spatiotemporal proximity for flight tracks in distributed MobilityDB is multidimensional tiling. Given the three-dimensional extent of the trips of the flight tracks, a three-dimensional grid of tiles is constructed to cover the entire spatiotemporal range of the trips [24]. Unlike hierarchical partitioning, where flight tracks are copied into all overlapping partitions, multidimensional tiling instead splits trips into segments at tile boundaries, ensuring that each tile only stored its respective segment. The grid-tiles are z-ordered and grouped into similar sized partitions while preserving the

spatial proximity of objects in each partition [23]. Those partitions can then be distributed among the *worker* nodes. Also multidimensional tiling allows the *coordinator* node of the MobilityDB cluster to generate distributed query plans that target specific partitions, and thus *worker* nodes holding these partitions [23].

Both hierarchical partitioning and multidimensional tiling have the potential to improve the performance of distributed MongoDB by enabling the spatiotemporal proximity of flight tracks. However, Citus does not natively support the algorithms required for creating these partitions and assigning them to *worker* nodes; instead, they must be implemented as custom SQL user functions [23]. Implementing one of these advanced sharding strategies could have enhanced the performance of MobilityDB's cluster setup while also improving its comparability to the distributed MongoDB setup. Spatiotemporal proximity is particularly crucial for the performance of spatiotemporal join operations, as the absence of proximity requires extensive data re-partitioning [23][22]. However, spatiotemporal JOIN operations were not employed in our queries due to MongoDB's limitations for spatiotemporal queries.

6 Related Work

To the best of our knowledge, we are the first to compare the performance of MobilityDB and MongoDB on 3D spatiotemporal data. However, there are a few approaches of benchmarking single-node as well as distributed setups of MongoDB and MobilityDB, as well as SECONDO and GeoMesa. In this section, we will guide you through some of them.

The BerlinMOD benchmark [31], published in 2009, is used to evaluate the performance of SECONDO. Beside a set of queries, BerlinMOD also provides a tool that generates moving point data. It simulates cars driving on Berlin's road network over a specified time period. A key advantage of using a data generator is its scalability—unlike real data, which is often limited in size, the generated data can be easily adjusted to match the benchmark requirements. The generator utilizes a graph representation of the street network. Nodes represent street crossings and dead ends, while streets are represented by edges between these nodes. The edges are assigned "costs" which symbolizes the time it takes to travel that street. BerlinMOD then models car trips by choosing start and end nodes on the network. The trips are modeled so that short-duration trips occur more frequently than long-duration ones [31], reflecting real-world travel patterns. Scaling the data increases the number of car trips and the number of simulation days [3]. The schema that is used by BerlinMOD includes data for regions, trips, cars, points, instants, and periods [3]. Furthermore, BerlinMOD comprises 17 queries that are based on that schema, including object-based, temporal, spatial, and spatiotemporal queries. All queries are executed on SECONDO to evaluate their performance across different data scale factors. Many of the queries from BerlinMOD have been adopted in subsequent benchmarks and also served as inspiration for some of our queries.

Zimányi et al. [3] utilize the BerlinMOD benchmark to evaluate the performance of MobilityDB and compare it to SECONDO. They demonstrate how queries designed for SECONDO, originally written in the SECONDO executable language, can be expressed in SQL for MobilityDB. Subsequently they apply these queries to SECONDO and MobilityDB across different scale factors of BerlinMOD's simulated car trip data. They report the average execution times of the 17 queries of which each is executed 5 times against MobilityDB and SECONDO. However, some of the queries are excluded in the benchmark as they crash or take too long to execute. The benchmark examines four different scale factors of the car trip data, resulting in a total execution time for all utilized queries of 5,700 seconds for MobilityDB and 7,300 seconds for SECONDO. While MobilityDB demonstrates an overall better performance, SECONDO outperforms it in some queries with lower average execution times. Zimanyi et al. conclude that, despite the higher stack of database mechanisms that MobilityDB builds on, the performance is rather good. Furthermore, they examine the impact of disabling MobilityDB's parallel query processing feature, thereby limiting the number of accessible cores [3]. Their findings show a significant performance gains when increasing from 1 to 2 cores or from 2 to 4 cores. However, beyond 6 cores, the performance improvement diminishes as queries become disk-bound and I/O becomes the primary bottleneck. In another study, Zimányi et al. report that for the BerlinMOD queries, when scaling car trips from 1,700 to 300,000, MobilityDB outperforms SECONDO in 63% of the queries. Moreover, the total execution time for all queries in MobilityDB amounts to just 13% of the time required in SECONDO [8]. However, these benchmarks comparing SECONDO and MobilityDB are conducted only on single-node setups and therefore do not provide insights into the systems' performance when scaling out to handle increased workloads.

In [35], Židić and Mastelić conduct performance and scalability benchmarks to compare the databases GeoMesa and SpaceTime. SpaceTime is a proprietary solution for storing and querying spatiotemporal data, requiring a solid-state drive (SSD) and a gigabit network for optimal performance. To evaluate both systems across different query types, the authors perform spatial, temporal, and spatiotemporal benchmarks. Additionally, to assess scalability, they execute spatiotemporal queries while varying the dataset size and the allocated resources of the database systems. The dataset used in the benchmarks consists of over a billion records of telecom data collected across Croatia over a 43-day period. Unlike most other spatiotemporal benchmarks, including ours, which primarily focus on trajectory data, this dataset consists of discrete point data. Both databases are deployed on two virtual machines each. GeoMesa uses Hadoop and Accumulo as the underlying storage layer. Each query is executed multiple times to ensure reliable average results. Spatial and temporal queries simply retrieve or count all records within a specified temporal or spatial range. Across all queried locations and time periods, SpaceTime outperforms GeoMesa. In the spatiotemporal benchmark queries fetch or count the records that occurred within varying timespans inside varying locations. SSpaceTime outperforms GeoMesa once again, showing a steady but moderate increase in response times as the queried data volume grows. GeoMesa's response time increases substantially when queries filter for smaller areas, despite returning the same amount of data as queries for larger areas. In contrast, SpaceTime does not exhibit this dependence, maintaining consistent response times regardless of the queried area's size. Moreover, SpaceTime demonstrates better scalability concerning data volume. However, its performance declines significantly when using hard disk drives (HDDs) instead of solid-state drives (SSDs). Additionally, SpaceTime benefits greatly from caching, meaning that reducing its available RAM leads to a noticeable drop in performance [35].

Makris et al. [6] evaluate and compare database systems used for spatiotemporal data analysis, specifically PostGIS, MobilityDB, and MongoDB. Their evaluation is based on a subset of queries from BerlinMOD, assessing the systems' ability to execute range, temporal aggregate, distance, and nearest-neighbor queries. The three systems are evaluated using real-world maritime data, specifically Automated Identification System (AIS) messages from 662 vessels in the Mediterranean Sea. In total, the dataset contains approximately 2,6 million spatiotemporal data points. However, the benchmark selects five queries from BerlinMOD, none of which are spatiotemporal; instead, they are either purely spatial or purely temporal. Overall, PostGIS delivers the best performance, with substantially lower average latencies for some queries. While MongoDB outperforms MobilityDB on three queries (two spatial and one temporal), MobilityDB excels in the distance and nearest-neighbor queries, which required custom function implementations in MongoDB. Moreover, Makris et al. performed another benchmark in which they segmented the vessels' trips. MobilityDB significantly benefits from trajectory segmentation, resulting in faster execution times for three out of the five queries. In another study, Makris et al. [21] evaluate MongoDB against PostgreSQL with the PostGIS extension using 11 spatiotemporal queries on a dataset containing vessel tracking points. Their evaluation includes executing queries with varying parameters, such as limiting the number of returned vessels or adjusting the time range of the temporal constraints in spatiotemporal queries. Furthermore, different polygons with different sizes are used for the benchmarks. The benchmarks are run on 5-node cluster setups of MongoDB and PostGIS that are deployed on Amazon Web Services (AWS) EC2 instances. The performance of both systems are measured in terms of response time of the queries with the changing parameters. They

show that PostgreSQL with the PostGIS extension outperforms MongoDB in almost all cases. The results further show that the use of indexes enormously reduces execution times for MongoDB, while the positive impact of using indexes on PostgreSQL is less profound [21]. However, in both clusters, the vessel tracking points are not sharded but instead replicated across all five nodes, meaning each node contains a full copy of the dataset. In contrast, we chose to shard the data across the clusters' nodes to evaluate how the databases handle an increasing workload on a large dataset.

Koutroumanis and Doulkeridis [4] evaluate MongoDB clusters utilizing different sharding strategies. They propose an indexing approach for spatiotemporal data that enables sharding which preserves spatiotemporal locality in shards. The baseline solution uses the same sharding strategy as our benchmarks, sharding documents by the date field and creating a compound index on the date and location fields. This approach only guarantees data locality at the temporal level. The approach they present is based on one-dimensional (1D) mapping of data using Hilbert space-filling curves [4]. For each document, a one-dimensional Hilbert value is determined based on the document's longitude and latitude. This value is then stored in a new field of type `Long`. The Hilbert index provides strong clustering and locality properties. The spatiotemporal shard key is subsequently defined as a compound index consisting of the Hilbert index field and the date field. Therefore, the sharding strategy exploits both, the spatial and the temporal dimension of the documents and therefore preserves spatiotemporal data locality. Documents that are spatially and temporally close are likely to be stored on the same *shard server*. The experiments, based on three spatiotemporal queries with varying spatial and temporal selectivity, yield mixed results in terms of execution time across the different indexing and sharding strategies. In some query configurations, the baseline index and sharding strategy perform better, while in others, the Hilbert index results in reduced execution times [4].

Bakli et al. [22] examine the scalability of MobilityDB using the Citus extension utilizing queries from the BerlinMOD benchmark. Four of the seventeen queries cannot be distributed by the *coordinator* node, as the Citus planner rejects them due to complex joins that are only supported when tables are joined on their distribution column. The remaining queries are evaluated on the BerlinMOD-generated car trip dataset across different scale factors ranging from 300,000 to 17 million car trips. Each query is executed five times, and the average response time is calculated. Bakli et al. conduct experiments on single-node setups and cluster setups with 4 and 28 nodes. In the 4-node setup, query response times are only a fraction of those in the single-node setup, with this fraction decreasing further as the scale factor increases. With 28 nodes, the difference in query response times between the cluster and the single-node setup becomes even more pronounced. However, the performance gain from scaling 4 to 28 nodes is not proportional to the cluster size increase, as even at the largest scale factor, the dataset remains too small to fully utilize a cluster of that size.

In another work, Bakli et al. [23] compare MobilityDB with other systems, such as SECONDO and Summit, in distributed setups comprising four physical machines. They evaluate performance using kNN queries, spatial range queries, temporal range queries, and spatiotemporal range queries on two datasets containing 90,000 and 227,000 ship trajectories, respectively. For distributed MobilityDB, they examine two partitioning methods: hierarchical partitioning and multidimensional tiling, which we introduced in Section **??**. Each query is executed five times, and the average query run-times are reported. For range queries, the authors systematically vary the query coverage across temporal, spatial, and spatiotemporal dimensions, with ranges spanning from 0.05% to 50% of the dataset's total

extent. Their approach follows a more methodical strategy by using predefined levels for temporal and spatial extents. However, by selecting spatial extents based on real-world regions, our benchmarks more closely resemble real-world scenarios. Across all range queries, irrespective of the covered range or partitioning method, MobilityDB consistently outperforms SECONDO by a small margin, while both systems demonstrate significantly better performance than Summit. Also for the kNN queries with varying numbers of examined trajectories, both MobilityDB performs substantially better than Summit for both partitioning methods. For assessing the scalability of distributed MobilityDB, Bakli et al. assess the effect of changing the number of *worker* nodes (from 5 to 30) and changing the dataset size. They test the performance of spatiotemporal range queries on the two different partitioning methods. The run-time performance for both partitioning methods improves linearly with the increase in the number of *worker* nodes, as local indexes now can fit into memory. Multidimensional tiling outperforms hierarchical partitioning, particularly for larger datasets and when the spatiotemporal range query covers large portions of the spatiotemporal data extent.

7 Conclusion

The widespread use of GPS tracking and IoT technologies has led to massive data collections on moving objects, requiring spatiotemporal databases to store this data and provide functionalities for applications to process and analyze it effectively [2]. This need is addressed by MobilityDB, an open-source extension for PostgreSQL and PostGIS that offers spatiotemporal data types and functions, thus supporting a variety of temporal, spatial, and spatiotemporal queries [2]. Furthermore, MobilityDB allows for distributed setups to handle the increasing amounts of spatiotemporal data [9]. However, beyond MobilityDB, databases that offer both extensive spatiotemporal querying capabilities and horizontal scalability are rare. MongoDB is a widely adopted NoSQL database that provides built-in support for geospatial data types and functions, enabling geospatial analysis. Additionally, it supports scalable cluster setups through sharding and replication [30].

To evaluate the suitability of the state-of-the-art databases MobilityDB and MongoDB as industry-scale spatiotemporal solutions, we benchmarked their performance using spatiotemporal workloads. Our evaluation comprised temporal, spatial, and spatiotemporal queries, assessing both systems in a single-node setup and a three-node cluster setup. The queries were designed to simulate real-world scenarios on a dataset containing three-dimensional flight tracks for approximately 1.8 million flights, along with geographic data describing regions within the evaluated area. However, MongoDB's support for spatiotemporal queries is limited. Therefore, we needed to ensure that our benchmarks included only queries that were feasible to implement on both SUTs while guaranteeing consistent results across both systems.

To run the benchmarks, we deployed both databases along with custom implementations of a *Benchmarking Client* on cloud infrastructure. Our experiments revealed substantial differences between the two systems in terms of query performance and scalability. Our results indicate that MongoDB outperforms MobilityDB in single-node setups, particularly for temporal and spatial queries, where MongoDB achieves substantially lower latencies. However, for spatiotemporal queries, MobilityDB performs slightly better than MongoDB. For an application-centric benchmark, the average latency across all query types in MobilityDB's single-node setup is 75% higher than that of MongoDB.

In distributed setups, both SUTs demonstrate improved performance, with increased throughput and reduced latencies in both the application-centric benchmark and the category benchmarks. However, MongoDB demonstrates superior scalability, handling a greater number of concurrent connections and delivering substantially higher throughput than MobilityDB. Furthermore, despite its higher throughput, MongoDB achieves lower latencies for spatial queries. The sharding strategy we employed for MongoDB enables efficient query routing, as temporal and spatiotemporal queries are directed only to the cluster nodes storing data for the respective temporal range. In contrast, under our sharding strategy for MobilityDB, the coordinator node broadcasts all queries to the worker nodes, regardless of query type. However, this limitation could be mitigated by implementing a more sophisticated sharding strategy for MobilityDB's distributed setup [23].

Partitioning and distributing data using hierarchical partitioning or multidimensional tiling could potentially enhance the performance of a MobilityDB cluster and represents a promising direction for future work. Furthermore, due to budget constraints, we relied on a single *Benchmarking Client* for evaluating both the single-node and distributed setups of

the SUTs. However, for more realistic benchmarks, future work should consider deploying multiple distributed database clients to better simulate real-world usage scenarios. Moreover, future benchmarks for MobilityDB and MongoDB could comprise a wider range of spatiotemporal queries, by implementing custom functions for MongoDB. Evaluating the performance of the SUTs under more complex query workloads, such as spatiotemporal joins, could provide deeper insights into their suitability for real-world applications. Additionally, the third dimension (altitude) is underrepresented in our benchmarks and should be more extensively incorporated into future benchmarks on 3-dimensional flight track data.

Abbreviations

ADT Abstract Data Type

AWS Amazon Web Services

BSON Binary JavaScript Object Notation

CRUD Create, Read, Update, Delete

CTE Common Table Expression

DBMS Database Management System

EC2 Elastic Compute Cloud

ETRS 89 European Terrestrial Reference System 1989

GCP Google Cloud Platform

GCE Google Compute Engine

GiST Generalized Search Tree

HDD Hard Disk Drive

HTTP Hypertext Transport Protocol

JSON JavaScript Object Notation

k-NN k-Nearest Neighbor

MOD Moving Object Database

NoSQL Not only SQL

NRW North Rhine-Westphalia

OGC Open Geospatial Consortium

SQL Structured Query Language

SP-GiST Space-partitioned Generalized Search Tree

SSD Solid State Drive

SUT System under Test

VM Virtual Machine

VPC Virtual Private Network

WGS 84 World Geodetic System 1984

YCSB Yahoo!Cloud Serving Benchmarking

References

[1] Andrew Hulbert, Thomas Kunicki, James N. Hughes, Anthony D. Fox, and Christopher N. Eichelberger. "An experimental study of big spatial data systems". In: *2016 IEEE International Conference on Big Data (Big Data)*. 2016, pp. 2664–2671. DOI: 10.1109/BigData.2016.7840909.

[2] Juan Godfrid, Pablo Radnic, Alejandro Ariel Vaisman, and Esteban Zimányi. "Analyzing public transport in the city of Buenos Aires with MobilityDB". In: *Public Transport* 14 (2022), pp. 287–321.

[3] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. "MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS". In: 45.4 (Dec. 2020). ISSN: 0362-5915. DOI: 10.1145/3406534.

[4] Nikolaos Koutroumanis and Christos Doulkeridis. "Scalable Spatio-temporal Indexing and Querying over a Document-oriented NoSQL Store". In: *International Conference on Extending Database Technology*. 2021.

[5] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, and Dimosthenis Anagnostopoulos. "Performance Evaluation of MongoDB and PostgreSQL for spatio-temporal data". In: Mar. 2019.

[6] Ioannis Kontopoulos, Antonios Makris, Stylianos Nektarios Xyalis, and Konstantinos Tserpes. "Benchmarking moving object functionalities of DBMSs using real-world spatiotemporal workload". In: *2022 23rd IEEE International Conference on Mobile Data Management (MDM)*. 2022, pp. 388–393. DOI: 10.1109/MDM55031.2022.00087.

[7] Open Geospatial Consortium. *OGC Moving Features*. Accessed: 2025-02-27. 2025.

[8] Esteban Zimányi, Mahmoud Sakr, Arthur Lesuisse, and Mohamed Bakli. "MobilityDB: A Mainstream Moving Object Database System". In: SSTD '19. Vienna, Austria: Association for Computing Machinery, 2019, pp. 206–209. ISBN: 9781450362801. DOI: 10.1145/3340964.3340991.

[9] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimányi. "Distributed Mobility Data Management in MobilityDB". In: *2020 21st IEEE International Conference on Mobile Data Management (MDM)*. 2020, pp. 238–239. DOI: 10.1109/MDM48529.2020.00052.

[10] David Bermbach. "Benchmarking Eventually Consistent Distributed Storage Systems". PhD thesis. 2014. 183 pp.

[11] Martin Grambow, Denis Kovalev, Christoph Laaber, Philipp Leitner, and David Bermbach. "Using Microbenchmark Suites to Detect Application Performance Changes". In: *IEEE Transactions on Cloud Computing* 11.3 (July 2023), pp. 2575–2590. ISSN: 2372-0018. DOI: 10.1109/tcc.2022.3217947.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st Symposium on Cloud Computing (SOCC)*. June 2010, pp. 143–154.

[13] Suneuy Kim, Yvonne Hoang, Tsz Ting Yu, and Yuvraj Singh Kanwar. "GeoYCSB: A Benchmark Framework for the Performance and Scalability Evaluation of Geospatial NoSQL Databases". In: *Big Data Research* 31 (2023), p. 100368. ISSN: 2214-5796. DOI: https://doi.org/10.1016/j.bdr.2023.100368.

[14] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. "A foundation for representing and querying moving objects". In: *ACM Trans. Database Syst.* 25.1 (Mar. 2000), pp. 1–42. ISSN: 0362-5915. DOI: 10.1145/352958.352963.

[15] Fabio Valdés, Thomas Behr, and Ralf Hartmut Güting. "Proposal for a Tutorial on Distributed Trajectory Management in Secondo". In: SpatialAPI'19. Chicago, IL, USA: Association for Computing Machinery, 2019. ISBN: 9781450369534. DOI: 10.1145/3356394.3365590.

[16] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. *Distributed SECONDO: an extensible highly available and scalable database management system*. en. Vol. 371. Informatik-Berichte. Hagen: FernUniversität in Hagen, 2016.

[17] Md Mahbub Alam, Luis Torgo, and Albert Bifet. "A Survey on Spatio-temporal Data Analytics Systems". In: 54.10s (Nov. 2022). ISSN: 0360-0300. DOI: 10.1145/3507904.

[18] GeoMesa Developers. *GeoMesa Documentation*. Accessed: 2025-02-11. 2024.

[19] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Yuan Sui, Jie Bao, and Yu Zheng. "TrajMesa: A Distributed NoSQL Storage Engine for Big Trajectory Data". In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 2020, pp. 2002–2005. DOI: 10.1109/ICDE48307.2020.00224.

[20] Sarthak Agarwal and K. Rajan. "Performance analysis of MongoDB versus PostGIS/PostGreSQL databases for line intersection and point containment spatial queries". In: *Spatial Information Research* 24 (Nov. 2016). DOI: 10.1007/s41324-016-0059-1.

[21] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, Dimitris Zissis, and Dimosthenis Anagnostopoulos. "MongoDB Vs PostgreSQL: A comparative study on performance aspects". In: *GeoInformatica* (Apr. 2021), p. 25. DOI: 10.1007/s10707-020-00407-w.

[22] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimanyi. "Distributed moving object data management in MobilityDB". In: *Proceedings of the 8th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. BigSpatial '19. Chicago, Illinois: Association for Computing Machinery, 2019. ISBN: 9781450369664. DOI: 10.1145/3356999.3365467.

[23] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimányi. "Distributed Spatiotemporal Trajectory Query Processing in SQL". In: SIGSPATIAL '20. Seattle, WA, USA: Association for Computing Machinery, 2020, pp. 87–98. ISBN: 9781450380195. DOI: 10.1145/3397536.3422262.

[24] MobilityDB Developers. *MobilityDB Documentation*. Accessed: 2025-02-11. 2024.

[25] Citus Data. *Citus Documentation*. Accessed: 2025-02-11. 2024.

[26] Veronika Abramova and Jorge Bernardino. "NoSQL Databases: MongoDB vs Cassandra". In: *Proceedings of the International C* Conference on Computer Science and Software Engineering*. C3S2E '13. 2013, pp. 14–22.

[27] Cornelia Győrödi, Robert Gyorodi, George Pecherle, and Andrada Olah. "A Comparative Study: MongoDB vs. MySQL". In: June 2015. DOI: 10.13140/RG.2.1.1226.7685.

[28] Longgang Xiang, Xiaotian Shao, and Dehao Wang. "PROVIDING R-TREE SUPPORT FOR MONGODB". In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLI-B4 (June 2016), pp. 545–549. DOI: 10.5194/isprs-archives-XLI-B4-545-2016.

[29] Anthony Fox, Chris Eichelberger, James Hughes, and Skylar Lyon. "Spatio-temporal indexing in non-relational distributed databases". In: *2013 IEEE International Conference on Big Data*. 2013, pp. 291–299. DOI: 10.1109/BigData.2013.6691586.

[30] Inc. MongoDB. *MongoDB Documentation*. Accessed: 2025-02-09. 2025.

[31] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. *BerlinMOD: a benchmark for moving object databases*. en. Vol. 340. Informatik-Berichte. Hagen: FernUniversität in Hagen, 2007.

[32] Christoph Laaber, Joel Scheuner, and Philipp Leitner. "Software microbenchmarking in the cloud. How bad is it really?" In: *Empirical Softw. Engg.* 24.4 (Aug. 2019), pp. 2469–2508. ISSN: 1382-3256. DOI: 10.1007/s10664-019-09681-1.

[33] PostGIS Development Team. *PostGIS Documentation*. Accessed: 2025-02-28. 2024.

[34] PostGIS Team. *PostGIS Workshop: Indexing*. Accessed: 2024-02-28. 2024.

[35] Dinko Židić and Toni Mastelić. *Comparing Open Source and Proprietary Database Solutions for Querying Spatio-Temporal Data: SpaceTime vs Geomesa*. Email: {dinko.zidic, toni.mastelic}@ericsson.com. Split, Croatia.

Hinweis zur Verwendung von Künstlicher Intelligenz bei der Textüberarbeitung

Zur Überarbeitung dieses Texte dieser Masterarbeit wurde KI verwendet. Der englische Abstract wurde mit DeepL ins Deutsche übertragen und anschließend überarbeitet. Der Haupttext wurde mit ChatGPT o3-mini-high auf Rechtschreibfehler, grammatische Unstimmigkeiten und unvollständige Sätze geprüft. Falls diese auftraten, wurden sie selbständig korrigiert.