Benchmarking the Scalability Behaviour of MobilityDB and CrateDB

Bachelor's Thesis

Author
Eryk Karol Kściuczyk
468796
e.ksciuczyk@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

Benchmarking the Scalability Behaviour of MobilityDB and CrateDB

Bachelor's Thesis

Submitted by:
Eryk Karol Kściuczyk
468796
e.ksciuczyk@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) Eryk Karol Kściuczyk, Berlin, 21. Juli 2025

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

Abstract

Kurzfassung

Contents

1 Introduction

Spatio-temporal databases are increasing in popularity due to increased production of such data by smart cities, personal light electric vehicles and numerous IoT devices. As the need for such databases raises, developers reach for production ready, scalable solutions. One of spatiotemporal databases is MobilityDB, built on top of the spatial extension PostGIS, extends PostgreSQL's capabilities regarding spatio-temporal aspects [ZSL20]. Previous research has shown that MobilityDB can also scale horizontally using another extension, Citus [BSZ19; BSZ20a; Cub+21]. In this thesis we will refer to distributed MobilityDB using Citus as MDBC . As a comparison, CrateDB is a natively distributable SQL database designed for scalability, high performance, and real-time applications. Both of the solutions are open source.

While the runtimes of specific queries of BerlinMOD benchmark by Düntgen, Behr, and Güting [DBG09] have been evaluated on 4 and 28 node clusters of MDBC by [BSZ20a], its horizontal and vertical scalability still remains underexplored. Previous research mostly focuses on designing queries for single node systems, and often does not compare systems between one another. The other system, CrateDB, also has not been fully evaluated for its usability in spatiotemporal use cases. Moreover, benchmarking with a wider range of cluster sizes is necessary to establish the scalability pattern. Additionally, a comparison between MDBC and CrateDB in terms of horizontal scalability has not been conducted, which could provide valuable insights into their respective strengths and weaknesses.

We therefore make the following contributions in this thesis:

1. We discuss the unique value propositions of both databases (Section 2)
2. We design and implement a benchmark of common functionality of both databases that addresses the scalability according to requirements we have identified (Section 3)
3. We setup and run the experiment on 4 cluster sizes using our custom benchmark (Section 4)
4. We analyze the results for scalability patterns and discuss the results of the benchmark (Section 4)

2 Background

2.1 Spatiotemporal Data and Queries

Spatiotemporal (ST) data contains spatial (location-based) and temporal (time-based) aspects. Geographic coordinates are a common way to make data geo-referenced, and a timestamp is often used as a temporal tag. [ATB22]

Furthermore, four types of spatiotemporal data can be distinguished. (i) **event data**, consisting of discrete occurrences at specific locations and times (e.g., sightings of wild animals in a national park), (ii) **trajectory data**, where trajectories of moving bodies are being measured (e.g., the route of a bus), (iii) **point reference data**, as described by [CW15], refers to measurements taken from a continuous ST process at specific locations that move over time (e.g., tracking surface temperature using weather balloons that drift through the atmosphere), and (iv) **raster data**, where collected observations of an ST field are gathered at fixed cells in an ST grid (e.g., fMRI scans of brain activity). [AKK18]

Stationary data refers to objects that do not change their position over time, such as weather stations and stationary sensors. Moving (dynamic) data refers to objects that change their position over time, such as vehicles, people, or animals.

There are many types of queries that can be performed on spatiotemporal data, here is the list of a common ones.

- **Spatiotemporal Range Query:** Retrieves objects or events that occurred within a specific geographic area during a defined time window. e.g. Find all vehicles that were in Berlin between 7:00 and 9:00 on Friday, June 6, 2025.
- **Time Slice Query:** Retrieve all objects that are in a specific spatial area at a single point in time. e.g. Find all buses that were in Warsaw in Wola district center at 15:00 on Tuesday, July 23, 2025.
- **Window Queries:** Retrieve all objects that intersect a spatial area during a given time window. e.g. Find all trains that passed through Berlin Central Station between 10:00 and 12:00 on Wednesday, May 4, 2025.
- **Trajectory Query:** Extracts or analyzes the movement path of an object over time. e.g. Show the movement path of a delivery drone as it traveled across Hamburg between 10:00 and 11:00 on Monday, June 2, 2025.
- **Spatiotemporal Join:** Identifies pairs of objects that came into spatial proximity during overlapping time intervals. e.g. Identify all pairs of taxis that came within 10 meters of each other between 16:00 and 20:00 on Friday night, June 6, 2025.
- **k-Nearest Neighbor:** Finds the closest objects to a given location at a specific time or within a time window. e.g. List the five ambulances that were closest to the intersection of Friedrichstraße and Unter den Linden in Berlin at 14:35 on Thursday, June 5, 2025.

2.2 Distributed Database Management Systems (DDBMS)

Distributed Database Management Systems (DDBMS) are systems in which the database is distributed across multiple physical locations, often connected via a network. These locations can be on different servers, in different data centers, or even across different geographic regions. The primary goal of a DDBMS is to manage a distributed database in such a way that it appears to the user as a single logical database, hiding the complexities of data distribution and replication. [OV96]

2.3 Challenges in Distributed Systems

Distributed systems, including DDBMSs, face several inherent challenges:

- **Network Latency:** Communication between nodes in a distributed system is significantly slower than accessing local memory or storage. This delay can affect query performance and data synchronization.
- **Data Consistency:** Ensuring that all nodes reflect the same data at the same time is difficult, especially in the presence of updates. Different consistency models (such as eventual consistency or strong consistency) are used depending on the application requirements.
- **Availability:** A highly available system must respond to requests even if some parts of the system are down. Maintaining availability while ensuring consistency can be challenging.
- **Partition Tolerance:** This refers to a system's ability to continue operating even when network partitions occur—i.e., when communication between some nodes fails. The CAP theorem (discussed below) emphasizes the trade-offs among consistency, availability, and partition tolerance.
- **Fault Tolerance:** Distributed systems must be resilient to hardware failures, crashes, and other unexpected issues. Mechanisms such as replication and failover are often used to ensure data is not lost and services continue to function.

2.3.1 CAP Theorem

A central concept in understanding distributed databases is the CAP theorem, introduced by [Bre00] and formally proven by [GL02], which states that a distributed system can guarantee only two of the following three properties at the same time:

- **Consistency (C):** Every read receives the most recent write or an error.
- **Availability (A):** Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- **Partition Tolerance (P):** The system continues to operate despite arbitrary message loss or failure of part of the system.

Due to the CAP theorem, distributed databases must make design trade-offs based on the requirements of the application they serve.

2.3.2 Handling Challenges in Different DDBMSs

Different distributed databases such as MDBC and CrateDB handle these challenges in different ways.

MDBC uses MobilityDB which is an extension of PostgreSQL that supports spatiotemporal data types and queries, commonly used in mobility and geographic information systems (GIS). To achieve horizontal scalability, MobilityDB is combined with Citus, another PostgreSQL extension that enables horizontal scaling through sharding and distributed query execution. Since MDBC is built on PostgreSQL and uses Citus for distribution, it maintains strong consistency and full ACID compliance. This is ideal for applications where data correctness and transactional integrity are critical.

CrateDB is a distributed SQL database designed for high ingest rates and real-time analytics over large-scale time-series and machine data. It is based on shared-nothing architecture using Elasticsearch and Lucene under the hood. CrateDB uses a distributed eventual consistency model, meaning data written to one node may take time to propagate to others. It relaxes ACID constraints to gain performance and scalability, making it less suitable for applications requiring strict consistency.

Additionally, the differences between MDBC and CrateDB can be seen by the available complex queries in MDBC compared to the possibilities of CrateDB. MDBC has comprehensive support for spatiotemporal queries, including complex queries such as spatiotemporal joins and k-nearest neighbor queries [BSZ20b][1]. On the other hand CrateDB focuses on the real-time analytics and horizontal scalability, which makes it a better solution for IoT devices and real-time analytics[2]. The table 2.1 shows the differences between MDBC and CrateDB and the queries the systems support.

Category	MDBC	CrateDB
Open source	Yes	Yes
Maintained by	PostgreSQL Global Development Group, Citus Data team (now part of Microsoft), MobilityDB Community, and OSGeo community	Crate.io
Consistency	Full ACID compliance	Eventual Consistancy
Supports SQL Queries	Yes	Yes
License	OSI-approved open source license similar to the BSD or MIT licenses	Apache 2.0
Supports spatiotemporal datatypes	Yes, builtin datatypes	Yes, geospatial data + time series (combined modeling)
Spatiotemporal Range Query	Yes	Partial (no native spatiotemporal types)
Time Slice Query	Yes	Yes
Window Queries	Yes	Yes
Trajectory Query	Yes	No
Spatiotemporal Join	Yes	Partial
k-Nearest Neighbor	Partial (spatial, some temporal)	Partial (spatial/vector, not spatiotemporal)

Table 2.1: Table comparing MDBC and CrateDB

[1] .

[2] .

2.4 Benchmarking

We designed a benchmark by using feature set supported by both systems under test (SUTs).

2.5 Cloud

3 Approach

Our benchmark is designed to investigate how CrateDB and MDBC compare in terms of scalability under spatio-temporal workloads typical for IoT use cases. Spatio-temporal aspects are the primary target of our benchmark, since MDBC value proposition is exactly that. Our goal is not to advocate for one system over the other but to investigate their respective strengths and limitations in processing spatio-temporal (ST) data at scale. To evaluate scalability, we have designed a benchmark seen in Figure 3.1.

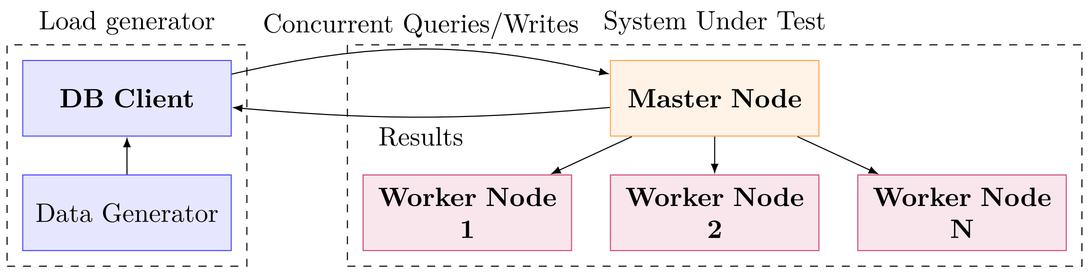

Figure 3.1: Load generator issues synthetic spatio-temporal inserts and queries to a SUT cluster consisting of one master and N worker nodes.

3.1 Infrastructure

We designed this benchmark to be deployed and run on the cloud for two reasons. First, we lacked access to multiple local machines with identical hardware, making resource heterogeneity a concern. Second, a cloud-based deployment improves repeatability, as others can reproduce our setup without dedicated hardware. Moreover, cloud deployments of distributed databases reflect real-world use cases, making the benchmark more relevant.

To improve reproducibility and repeatability, all parts of the benchmark, including cloud resource definitions and software configuration, are expressed in code. Using infrastructure as code allows multiple benchmark runs with the exact same configuration eliminating the risk of manual misconfiguration. Additionally, systems having access to the identical resources makes the benchmark and comparison fair.

3.2 Load generator

We decided to split the benchmark into two major components, the DDBMS cluster and a load generator. The load generator simulates concurrent clients on a DDBMS cluster. It uses a synthetic micro benchmark workload with fixed user pool. Simulating multiple clients at the same time strives to mimic the IOT workload where there are many devices issuing requests at the same time. The data for the inserts is generated before the load generator runs, and the queries are generated on runtime.

By using a micro-benchmark workload, we sacrifice the realism of the application to improve understandability, simplify the development, and interpretability of the results. We chose synthetic workloads to isolate system behavior and control variability by issuing similar queries and inserts. While this reduces workload realism, it improves reproducibility and allows clearer attribution of performance effects.

Load generator is deployed in the cloud as well, close to the SUT instead of running it locally and sending queries through the internet. This approach allows us to minimize latency and potential disturbances while the benchmark runs i.e., packet loss due to network problems.

3.3 DDBMS Clusters

Each DDBMS is deployed on a cluster of machines with the same specifications that meet or exceed each system's minimum requirements. This allows fair comparison of both systems as they have access to same resources. The infrastructure of the cluster is deployed in the cloud using infrastructure as code and the DDBMS is deployed on the infrastructure using an automation tool. Using an automation tool allows repeatable deployment of the DDBMS in the same way as infrastructure as code allows repeatable deployment of the resources.

3.4 Workloads and metrics

During the benchmark run, the load generator measures the following metrics for the analysis afterwards: (1) Latency - measured in milliseconds between sending the request to DDBMS and receiving a response from SUT, (2) Success rate - this allows us to avoid a situation where one SUT would perform better by rejecting multiple queries, (3) Throughput, by logging how many queries have been sent and the total time it took to do it, we calculate the throughput.

Latency and throughput were selected as primary metrics due to their relevance in real-time and high-volume IoT scenarios. A system is considered to scale effectively if throughput increases sub-linearly or better with cluster size, and latency remains within acceptable bounds under increasing load.[HHK23] Furthermore, throughput is important for IoT deployments as they are commonly deployed in large quantities and send a numerous inserts i.e., e-scooters deployed in a city which report their location and status.

Moreover, we have identified three scenarios for the SUT which we plan to benchmark: (1) Data insertion, a scenario where the clients send data to insert to the DDBMS. By this we want to benchmark how the SUT performs when multiple clients insert data i.e., e-scooters deployed in a city reporting their status and location. Here we find the write throughput metric extremely relevant. (2) Simple read queries, in this scenario we try to simulate queries done by the users of a system i.e., people using the e-scooter app to find the closest e-scooters. In this scenario we benchmark the read throughput, as it is a relevant metric for the clients of the system. (3) Complex read queries, here we try to simulate a scenario of a complex queries used for analysis and optimizations i.e., queries done by the e-scooter company to find patterns in the traffic to optimize the placement of the e-scooter stations throughout a city or to gain insight into their data.

3.5 Scalability

To establish a scalability pattern we vary cluster sizes to evaluate the horizontal scalability. Each cluster size is benchmarked using several user pool sizes, simulating different concurrency levels. Benchmarking the DDBMS using different client counts allow us to check whether the performance of the queries is improved, such as the response time, or the amount of simultaneous connections handled with acceptable latency. This may lead to important findings as some companies may value the performance of the queries for their data analysis more than concurrent connections handled. On the other hand a different company may prioritize performance for numerous concurrent devices i.e., an e-scooter company having a large fleet of vehicles.

Through this benchmark design, we aim to uncover scalability patterns of two distributed databases under realistic IoT-inspired workloads. The evaluation results will inform system design choices for practitioners building large-scale spatio-temporal data processing pipelines.

4 Experiment

4.1 Load Generator

This approach allows easy data collection, compared to using multiple load generators running on multiple hosts and makes the benchmark cheaper to run. We believe this strategy is viable, as the load generator primarily issues requests, which are i/o heavy, so the performance of the computer running the software shouldn't be a bottleneck. The load generator is also put on the cloud in the same virtual private cloud (VPC) as the DDBMS cluster. Putting the load generator next to the SUT instead of running it on our local computer allows us to minimize latency and potential disturbances while the benchmark runs i.e., packages dropped due to network problems.

4.2 DDBMS cluster

We document the DDBMS cluster configuration through Kubernetes deployment files ensuring that the software will be deployed with the same settings and versions using versioned container images. Additionally by using Kubernetes, the benchmark can also be more easily developed on local systems by using contenerization. The local development has been an important consideration as the recent withdrawal of Google from issuing cloud tokens to our university, lead to uncertainty to where the benchmark will be deployed. Kubernetes has been chosen over using bash scripts for installing DDBMS on multiple virtual machines for two reasons. (1) It allows easier deployment of a DDBMS cluster, which is beneficial when deploying multiple cluster sizes and resetting their state by destroying them. (2) Relevancy, as deployment of such clustered DDBMS systems is often done this way in production environments (CITATION HERE)

In the benchmark the load generator will connect with the System Under Test (SUT) and issue queries. It will log metrics allowing later analysis of the findings. The System Under Test, will be run on the same resources i.e., MobilityDB will be deployed on the same resources as CrateDB. The resources for the SUT will be restarted between each benchmark run to ensure a clean state.

4.3 Design Objectives

The benchmark should be relevant to real-world use cases, especially spatial-temporal workloads as this is the primary value proposition of MobilityDB over other databases. We accomplish that by modelling queries after common use cases like, time slice queries, window queries, and spatiotemporal joins. We deploy the SUTs on a Kubernetes cluster in Microsoft Azure, resembling common deployment method of companies. Such deployment using infrastructure as code, allows us to share the configuration of SUTs, as well as information of provisioned cloud resources, allowing reproducibility of our results.

The decision to choose Kubernetes over shell scripts or other automation tools such as Ansible has reduced understandability (as not everybody is familiar with container orchestration tools such as Kubernetes), but allowed us to improve portability, relevance and simplified development. This trade off has been made as unexpectedly because of the cloud infrastructure provider, where we wanted to run the benchmark on initially, cut off credits for the university right before the start of the thesis writing. Such change, made us switch to a solution that was more portable. Using containerization, allowed us to develop on single host machine without creating VMs and also made it possible to run our benchmark on other cloud providers or university servers/Kubernetes cluster. Furthermore, use of containers improves repeatability as the node will have exact same versions of the software installed on them.

The SUTs are benchmarked using equivalent conditions ensuring fairness by (1) deploying them on the same cloud resources (processor types, memory, storage, and networking), (2) running semantically analogous queries, (3) connecting same amount of clients to them, and by (3) inserting same amount of data. Notably, MDBC supports more complex queries than CrateDB and because of that we will benchmark queries supported by both systems. We suspect that MDBC being full ACID compliant will result in reduced raw performance, but in this thesis we focus on scalability patterns so are making this trade-off to fairness.

Our benchmark evaluates scalability by measuring how each SUT performs under growing cluster sizes and data volumes.

We decided to benchmark the following sizes for the SUT clusters, 2, 3, 4, and 5. Preferably we would benchmark the DDBMS on larger cluster sizes to improve relevance of our findings, but due to resource constrain we choose to do it on a set of small ones to reduce costs. Despite the small difference in size between the sizes, we hope to see a pattern which would allow us to create conclusions about scalability.

To check how well the databases handle multiple simultaneous connections and try to find a pattern, we have decided to benchmark the SUT on following number of simultaneous clients 100, 1000, 10000.

4.4 Quality Metrics

To measure scalability we plot the Metric curves over increasing node counts. We choose the following metrics for our benchmark (1) Latency, it will be measured in milliseconds for both inserts and queries. We will create percentile breakdown (P50,P95, P99) to provide insight into tail latencies. (2) Throughput, we will measure the number of records written or queries completed per second.

In the context of IoT devices, write throughput is important for the devices themselves, and the read latency is important for the users. Because of this, the benchmark results should be relevant.

4.5 Workload design

We adopt a synthetic workload combined with a micro-benchmark approach where each run isolates one quality of system behavior (e.g., write latency, read throughput). This avoids confounding metrics and ensures interpretable results. We have developed a custom synthetic workload generator, written in Golang, which will issue concurrent requests using lightweight goroutines to simulate a fixed large number of clients. Golang has been chosen due to goroutines, which are lightweight threads (2KB stack vs 1MB stack when using OS threads) managed by the Go runtime, allowing us to simulate a large number of concurrent clients on a relatively small system.

For the write workload we will simulate IoT devices emitting time-stamped spatial data of We will test batch and single-record inserts at varying ingestion rates to measure write throughput and latency.

Whereas for the read workload, we will simulate This will be performed by running following queries (1) Temporal range queries (e.g., last hour of data per device). (2) Spatial bounding box queries (e.g., devices within a bounding box). (3) Spatio-temporal window queries (e.g., average speed over 10-minute intervals).

4.6 Measurement

We combined the load generator with the testing client in order for the same goroutine to issues requests as well as to measure their metrics. This way we keep the measuring simple and understandable and don't overcomplicate the setup.

In order to minimize unknown variables, such as network latency, reliability, and interference of other devices, we put the load generator on the same Kubernetes cluster. The load generator pod runs on a separate Kubernetes node, a different host machine, isolated from database nodes to prevent influencing SUT performance. During the evaluation we monitor the resource usage of every node, focusing on the load generator in order to prevent it from becoming a bottleneck in the benchmark.

5 Discussion

The benchmarked cluster configurations were limited, clusters of bigger sizes should also be evaluated, such as cluster of size 40. We were not able to do it due to limited resources and lack of Google Cloud Platform tokens, which have not been issued to the University recently.

One could also run the benchmarks over a larger time frame i.e., running the benchmarks every day or week on different times for a few months. This would reduce the variability of the cloud providers systems, as previous research has demonstrated that the time of the day can impact performance of virtual machines hosted on the cloud. (ADD CITATION)

Future research could also look into whether change in the hardware of each node would play a role in scalability i.e., whether improving the hardware of each node in the cluster would influence the scalability patterns.

Wider range of concurrent client connections to the systems could be benchmarked in order to test how do the systems handle even large amount of simultaneous clients. This would additionally lead to more reliable findings, as the amount of interpolation between the performance points would be smaller.

Due to time constraint, we additionally were not able to evaluate performance of different query types, such as updates and deletes.

Additionally, the vertical scalability of each system has not been determined and compared.

Furthermore, a more realistic workload could be benchmarked, such as a trace based one instead of micro-benchmark.

The systems would also need to be evaluated when nodes of the DBMS would be running on bare metal servers, not on top of Kubernetes inside docker containers as containerization alongside the Kubernetes overhead could impact the results.

Consistency has not been evaluated, as MDBC supports ACID compliant transactions and CrateDB guarantees only eventual consistency. Future research could check how much time does CrateDB need to get into consistent state. One could also evaluate the performance and scalability of MDBC with transactions and without them.

6 Related Work

7 Conclusion

References

[] *CrateDB Documentation.* https://cratedb.com/docs/guide/home/index.html.

[] *Querying the Data MobilityDB Docs.* https://docs.mobilitydb.com/MobilityDB-BerlinMOD/master/ch01s05.html.

[AKK18] Gowtham Atluri, Anuj Karpatne, and Vipin Kumar. "Spatio-Temporal Data Mining: A Survey of Problems and Methods". In: *ACM Comput. Surv.* 51.4 (Aug. 2018), 83:1–83:41. ISSN: 0360-0300. DOI: `10.1145/3161602`.

[ATB22] Md Mahbub Alam, Luis Torgo, and Albert Bifet. "A Survey on Spatio-temporal Data Analytics Systems". In: *ACM Comput. Surv.* 54.10s (Nov. 2022), 219:1–219:38. ISSN: 0360-0300. DOI: `10.1145/3507904`.

[Bre00] Eric A. Brewer. "Towards Robust Distributed Systems". In: *PODC*. Vol. 7. Portland, OR, 2000, pp. 343–477.

[BSZ19] 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.* Chicago Illinois: ACM, Nov. 2019, pp. 1–10. ISBN: 978-1-4503-6966-4. DOI: `10.1145/3356999.3365467`.

[BSZ20a] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimanyi. "Distributed Mobility Data Management in MobilityDB". In: *21st IEEE International Conference on Mobile Data Management (MDM).* June 2020.

[BSZ20b] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimányi. "Distributed Spatiotemporal Trajectory Query Processing in SQL". In: *Proceedings of the 28th International Conference on Advances in Geographic Information Systems.* SIGSPATIAL '20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 87–98. ISBN: 978-1-4503-8019-5. DOI: `10.1145/3397536.3422262`.

[Cub+21] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. "Citus: Distributed PostgreSQL for Data-Intensive Applications". In: *Proceedings of the 2021 International Conference on Management of Data.* Virtual Event China: ACM, June 2021, pp. 2490–2502. ISBN: 978-1-4503-8343-1. DOI: `10.1145/3448016.3457551`.

[CW15] Noel Cressie and Christopher K. Wikle. *Statistics for Spatio-Temporal Data.* John Wiley & Sons, Nov. 2015. ISBN: 978-1-119-24306-9.

[DBG09] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. "BerlinMOD: A Benchmark for Moving Object Databases". In: *The VLDB Journal* 18.6 (Dec. 2009), pp. 1335–1368. ISSN: 1066-8888, 0949-877X. DOI: `10.1007/s00778-009-0142-5`.

[GL02] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: `10.1145/564585.564601`.

[HHK23] Tobias Hossfeld, Poul E. Heegaard, and Wolfgang Kellerer. "Comparing the Scalability of Communication Networks and Systems". In: *IEEE Access* 11 (2023), pp. 101474–101497. ISSN: 2169-3536. DOI: `10.1109/access.2023.3314201`.

[OV96] M Tamer Ozsu and Patrick Valduriez. "Distributed and Parallel Database Systems". In: *ACM Computing Surveys* 28.1 (1996).

[ZSL20] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. "MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS". In: *ACM Transactions* *on Database Systems* 45.4 (Dec. 2020), pp. 1–42. ISSN: 0362-5915, 1557-4644. DOI: `10.1145/3406534`.