UNIVERSITY OF COPENHAGEN

PhD in Computer Science

Scalable and Reactive Data Management for Mobile Internet-of-Things Applications with Actor-Oriented Databases

Yiwen Wang

Supervised by Marcos Antonio Vaz Salles

April 2021

Yiwen Wang

Scalable and Reactive Data Management for Mobile Internet-of-Things Applications with Actor-Oriented Databases

PhD in Computer Science, April 2021

Supervisors: Marcos Antonio Vaz Salles

University of Copenhagen

Faculty of Science

PhD Degree in Computer Science

Sigurdsgade 41

2200 Copenhagen

四载春秋，畏作黄粱，虽非寒窗，亦是苦读。知己无柳絮高才，幸有大贤焉而为其徒，博学慎思，人十能之而己百之，笃行亦则足恃矣。今停笔止言之际，回首旦暮，赠以诗酒共年华，赚得知交满天下。愿今后去往之地，皆为热土，期明朝漫漫修远之路，皆伴长风济沧海。

Acknowledgements

My PhD work conducted in the context of the Future Cropping partnership (*Future Cropping partnership website* 2018), supported by Innovation Fund Denmark. Experimental evaluation partially supported by the AWS Cloud Credits for Research program. In addition, this work was partly supported by the International Network Programme project "Modeling and Developing Actor Database Applications", funded by the Danish Agency for Science and Higher Education (number 7059-000528) and by FAPESP CEPID CCES 13/08293-7. Additional funding provided by FAPESP project 17/02325-5 and by CNPq-Brazil, Department of Computer Science, University of Copenhagen and Programming technology foundations for Accountability, Privacy-by-design & Robustness in Context-aware Systems (case number: 9131-00077B).

Throughout the working of this nearly four years PhD pursuing adventure, I have received a great deal of support and assistance from many aspects. First and foremost, I would like to express my special appreciation to my supervisor, Professor Marcos Antonio Vaz Salles, whose expertise was invaluable in guiding me in the research. I would like to thank you for your insightful feedback pushed me to sharpen my thinking as a researcher and brought my work to a higher level. During the special time when the world been attacked by Covid-19, everything has become more complicated and challenging. Thank you for providing me supports in all aspects and even arrange time to help me every week during your parental leave and vacations. I have deep gratitude for that in my heart. You are the best supervisor in the world.

Meanwhile, I would like to thank Professor Yongluan Zhou, Professor Claudia Bauzer Medeiros, Professor Julio Cesar Dos Reis, Vivek Shah and Kasper Myrtue Borggrens, for their extraordinary cooperation, inspirations, insightful comments, and suggestions during my PhD study. Special thanks to Professor Christian S. Jensen for hosting me at Aalborg University for my environmental

exchange. I would like to acknowledge my colleagues in Sigurdsgade 41 for our discussion, brainstorming, and wonderful joy time.

I would also like to thank all my friends. I would like to offer my special thanks to Ziming Luo, who has provided company for me from high school to university, and also in my current PhD study. Thank you for providing happy distractions to rest my mind outside of my research, cooking delicious food, and taking care of my pet fish, Panda.

I would particularly like to thank my beloved husband. I cannot thank you enough for encouraging me throughout my PhD experience. I really appreciate your wise counsel and sympathetic ear. You are the victim of my crying, screaming, disordering, and weird behaviors in my stressed time. Meanwhile, I would like to thank my lovely family-in-law for embracing me with the family atmosphere when I am thousands of kilometers away from my hometown.

Finally, I could not have completed my PhD study without the support of my parents, my sister, and my deceased grandparents. Thank you for your unwavering support and belief in me. I know you are always there for me. Also, I think this dissertation is a perfect prenatal education material for my unborn niece.

Abstract

Development and utilization of Internet-of-Things (IoT) applications are increasing at an unprecedented speed in many domains, such as supply chain and logistics tracking, smart agriculture, e-health, and intelligent transportation systems, to name a few. Among these applications, mobile IoT applications have received special attention by virtue of having an ever more significant impact on people's day-to-day and society. Mobile IoT applications employ widely used portable and movable IoT entities that collect data, share information, and interact with each other to achieve common goals. While a host of data management solutions may be employed in the architecture of IoT applications, ranging from edge data processing to historical analytical database systems, a particularly interesting component is the IoT data platform, which is a cloud-resident infrastructure for online management of IoT data.

The growing connection coverage and increasing requirements in mobile IoT applications bring about special problems and challenges to be explored in IoT data platforms. Firstly, scalability is a necessary requirement for an IoT data platform because of the explosive growth of the IoT. Secondly, tight low-latency reactivity in an IoT data platform is required while managing a massive amount of highly concurrently generated data. Thirdly, support for heterogeneous data types is demanded in an IoT data platform due to the variety of IoT devices. Fourthly, elasticity is also required in an IoT data platform to handle dynamically changing IoT workloads. Last but not least, guidance on ensuring the dynamic and flexible development of IoT data platforms is crucial for application developers.

Existing approaches have explored how to support different properties required in IoT data platforms, but they fall short of fulfilling our goal of providing programmable, scalable, and reactive data management for mobile IoT applications. Recently, Actor-Oriented Databases (AODBs) have been proposed

as a new cloud-based data management architecture for distributed, scalable, stateful, and interactive applications. We find that AODBs are naturally suitable for satisfying the requirements and solving the challenges of building IoT data platforms. Therefore, we inspect the distinct requirements of building IoT data platforms through two case studies, and we investigate how to effectively model and build IoT data platforms with AODBs. Then, we focus on easing the complexity of building mobile IoT data platforms with AODBs while providing scalability and reactivity. We propose a novel class of data management systems called Moving Actor-Oriented Databases (M-AODBs), which integrate the new abstraction of moving actors for reactive moving objects with two data concurrency semantics for different application use scenarios. A first implementation, Dolphin, is presented to validate the design of M-AODBs. Afterward, to handle the typically skewed spatial distributions in mobile IoT applications, we apply varied classic spatial partitioning techniques in Dolphin to evaluate, analyze, and manage bottlenecks due to data skew. Our experimental study illustrates the impact of spatial partitioning in Dolphin and unveils several promising directions for future work.

Resumé

Udvikling og anvendelse af Internet-of-Things (IoT) applikationer sker med en hidtil uset hastighed på flere anvendelsesområder, såsom forsyningskæde og logistiksporing, smart landbrug, e-sundhed og intelligente transportsystemer. Blandt disse applikationer har mobile IoT-applikationer fået særlig opmærksomhed i kraft af at have en stadig større indflydelse på samfundet og folks dagligdag. Mobile IoT-applikationer anvender meget bærbare og bevægelige IoT-enheder, der indsamler data, deler information og interagerer med hinanden for at nå et større fælles mål. Mens en række datastyringsløsninger kan anvendes i arkitekturen af IoT-applikationer, der spænder fra kantdatabehandling til historiske analytiske databaser, så er en særlig interessant komponent IoT-dataplatformen, som er en cloud-resident infrastruktur til online styring af IoT-data.

Den voksende forbindelsesdækning og stigende krav i mobile IoT applikationer medfører specielle problemer og udfordringer, der skal udforskes i IoT-dataplatforme. For det første er skalerbarhed et nødvendigt krav til en IoT-dataplatform på grund af den eksplosive vækst i IoT. For det andet kræves en lav-latency-reaktivitet i en IoT-dataplatform, når der skal administreres en massiv mængde stærkt samtidig genererede data. For det tredje kræves det at heterogene datatyper understøttes i en IoT-dataplatform på grund af forskelligheden blandt IoT-enhederne. For det fjerde kræves elasticitet i en IoT-dataplatform for at kunne håndtere dynamisk skiftende IoT-arbejdsbelastninger. Sidst men ikke mindst, så er der behov for vejledning til at sikre en dynamisk og fleksibel udvikling af IoT-dataplatforme for applikationsudviklere.

Eksisterende løsninger har udforsket, hvordan de forskellige egenskaber der kræves i IoT-dataplatforme kan understøttes, men opfylder ikke vores mål om at sikre programmerbar, skalerbar og reaktiv geodatastyring til mobile

IoT-applikationer. For nylig er Actor-Oriented Databaser (AODBs) blevet foreslået som en ny skybaseret datastyringsarkitektur for distribuerede, skalerbare, stateful og interaktive applikationer. Vi finder, at AODBs naturligvis er velegnet til at opfylde kravene og løse udfordringerne der er relateret til opbyggelsen af IoT-dataplatforme. Derfor undersøger vi de specifikke krav der er relateret til opbygningen af IoT-dataplatforme gennem to casestudier, samt hvordan man kan effektivt modellere og bygge IoT-dataplatforme med AODBs. Derefter fokuserer vi på at lette kompleksiteten relateret til opbyggelsen af mobile IoT-dataplatforme med AODBs, mens vi samtidig sikre skalerbarhed og reaktivitet. Vi foreslår en ny klasse af datahåndteringssystemer kaldet Moving Actor-Oriented Databases (M-AODBs), der integrerer den nye abstraktion af moving actors til reaktive bevægende objekter med to datasamtidige semantikker til forskellige applikations scenarier. Den første implementering, Dolphin, præsenteres for at validere designet af M-AODBs. Bagefter anvender vi forskellige klassiske partitioneringsteknikker i Dolphin til at evaluere, analysere og styre flaskehalse på grund af dataforskydning for efterfølgende at kunne håndtere de typisk ikke-ensartet spatialle fordelinger i mobile IoT-applikationer. Vores eksperimentelle undersøgelse illustrerer virkningen af rumlig partitionering i Dolphin og afslører flere lovende retninger for fremtidigt arbejde.

Contents

1 Introduction

1.1 Motivation

1.1.1 Background

Deployment of the Internet-of-Things (IoT) has seen unprecedented growth in many aspects of today's society (Fizza *et al.*, 2021; Lee and Lee, 2015; Kraijak and Tuwanut, 2015). IoT consists of a diverse range of interconnected entities, such as sensors, mobile phones, appliances, and actuators, that collect data, share information, and interact with each other. IoT entities connect through the Internet to achieve common goals (Gubbi *et al.*, 2013). The IoT enables users and devices to become more informed and informative through sharing information and it facilitates many current and emerging applications in various areas, e.g., supply chain and logistics tracking (Sørensen and Bochtis, 2010; Burgard *et al.*, 2000), intelligent transportation (Papp *et al.*, 2008; Li and Nashashibi, 2013; Hu *et al.*, 2020; Tak *et al.*, 2020), smart agriculture (Jeppesen *et al.*, 2018; *Agricultural Robotics: The Future of Robotic Agriculture* 2020; Albani *et al.*, 2017; McLellan, 2020), hazard monitoring (Paul and Sarath, 2018; Qin *et al.*, 2018), to name a few. Meanwhile, IoT applications include assisted living and e-health (Minoli *et al.*, 2017; Javaid and Khan, 2021; Bhowmick *et al.*, 2021), IoT social network enhancement (Atzori *et al.*, 2012), augmented reality (Billinghurst *et al.*, 2015; Gutierrez *et al.*, 2012), and smart building and smart city (Arasteh *et al.*, 2016; Flores-Martin *et al.*, 2021), etc.. IoT applications are subject of lively debate as they increasingly permeate today's life. IoT devices are also increasing in quantity, density, diversity, as well as becoming critical for end users (Nahrstedt *et al.*, 2016). Unquestionably, as a concept that has been actively discussed and developed in many areas for

many years, the IoT is having an ever greater impact on people's day-to-day and behavior and represents a window of opportunity for business, industry, and academic research.

In an IoT system, we find IoT not only interconnects static devices that are built into unmovable infrastructures. Mobile IoT is also incorporating spatial flexibility, through, e.g., vehicles, wearable devices, and mobile phones. Mobile IoT applications include intelligent transportation systems (ITS) (Muthuramalingam *et al.*, 2019; Anand *et al.*, 2015; Tak *et al.*, 2020), augmented maps (Woods, 2021), location-based recommendation (Ojagh *et al.*, 2020; Chen *et al.*, 2017), and restraining order violation monitoring (Abd El-Aziz *et al.*, 2012; Elrefaei *et al.*, 2017; Tundis *et al.*, 2020), to mention a few. The differences between mobile vs. traditional IoT applications include: 1) mobility, in that mobile IoT applications are usually deployed on portable devices that can move around in space (Nahrstedt *et al.*, 2016); 2) Internet access and connection, in that mobile IoT applications often connect to the Internet through wireless networks due to their mobility (Srinivasan *et al.*, 2019); 3) energy availability, in that due to the relatively small size of batteries and bounded energy density, available energy in mobile IoT devices is limited (Pasricha *et al.*, 2020); and 4) security and privacy, in that mobile IoT devices dynamically connect with different entities and exchange device identifiers, but must preserve anonymity and privacy of users (Sharma *et al.*, 2020). Among those differences, mobility becomes the first property that separates mobile from traditional IoT, and the explosion of the mobile IoT has helped increase the generation of spatial data (Iyer and Stoica, 2017).

To be noticed, investigations of mobile IoT and its interaction with the surrounding environment have been conducted for many years, such as in ITS (Li and Nashashibi, 2013; Hu *et al.*, 2020). However, the growing connection coverage of mobile IoT and increasing reactive cooperation requirements in the IoT bring new challenges. For example, regular ITS collect information from vehicles and utilize received information to conduct optimization and coordination on system level. By contrast, in cooperative intelligent vehicle systems (C-ITS) (Mitsakis *et al.*, 2020; Bussche, 2020; Nguyen *et al.*, 2020; Uhlemann, 2018), vehicles are more than information providers, actively and directly participating in information exchange and decision making. Cooperation among vehicles in C-ITS is a crucial feature that assists in driving behaviors and enhances transportation effectiveness, security, and safety (Ni, 2016). All of these application scenarios make mobile IoT applications an

integral part of the fabric of IoT applications that brings about special problems and challenges.

1.1.2 Problem and Challenge Statements

The explosion of the IoT and its applications brings not only potential benefits, but also engenders the complexity of IoT systems. IoT systems include hardware systems such as IoT devices and physical transmission media as well as software systems such as applications software and data management platforms. In our work, we exclusively focus on the data and data management software modules that are part of an IoT system, and leave the entire IoT software ecosystem and hardware systems involved in an IoT scenario out of our discussion.

There are numerous data management challenges to be explored to realize the full potential of IoT applications, which contain broad areas that cover many systems and stakeholders. For instance, in-device data management and edge computing in IoT ecosystems; data management on historical data for analytical databases; privacy, security, and trust issues; quality of IoT service and management of massive datasets. While IoT applications provide valuable insights in varied aspects, online IoT data processing provides time-sensitive insights with high potential impact. Therefore, our work focuses on IoT data platforms, including online data processing, querying, updating, and reactivity.

The problems of IoT data platforms arise from the following aspects and are accompanied by challenges that need to be solved. First and most basically, an IoT data platform needs to be capable to scalably manage online IoT data due to the explosive growth of IoT. Scalability for IoT data platforms must not only consider data volumes, but also the speed of data flow and the handling of data streams from a multitude of devices.

Second, it is necessary to facilitate reactivity to incoming IoT data with low latency. Many IoT systems require processing of a massive amount of highly concurrently generated data. However, how to achieve the management of potential interactions among data to enable reactive behaviors with low latency remains an open research problem.

Third, mobile IoT applications can be arbitrarily distributed across space, and data from varied IoT devices have heterogeneous structure. It is usually

infeasible to store data from IoT in a single relational table with a fixed format. Therefore, IoT data management needs to be flexible and adapt to handling increasingly heterogeneous IoT data while guaranteeing data protection from different entities.

Fourth, the existence of IoT entities is also dynamic. In many IoT scenarios, it is common for devices to join and leave the system over time. Therefore, the elasticity of data management systems is also required.

Last but not least, there is a lack of support to application developers for the construction of an IoT data platform. Because there is no standard functionality that is logically defined for IoT applications, the data platform functionality heavily depends on the given application domain, and IoT data platform developers need to take on the burden of building functionality while considering associated data management. There is an absence of a consolidated way of how to design and develop an IoT system. Therefore, when focusing an investigation on how to manage the data from large volumes of devices, at the same time ensuring the dynamic and flexible development of applications is also a significant challenge of an IoT data platform.

Currently, there is a lack of data management frameworks to scalably manage mobile IoT data while achieving low-latency reactions in potentially complex software applications, as we discuss further in the following section.

1.2 State of the Art

In this section, we introduce existing popular approaches that are related to our work. These approaches have explored how to support different properties required in building reactive mobile IoT data platforms. These systems employ a variety of data-centric system abstractions. We illustrate their advantages and limitations and explain why it is necessary to explore a new system to solve all the issues identified.

1.2.1 Spatial Data Management Systems

Given the importance and broad relevance of spatial data from mobile IoT applications, firstly, we investigate database systems that support spatial services. There is an extensive range of spatial data management solutions that we

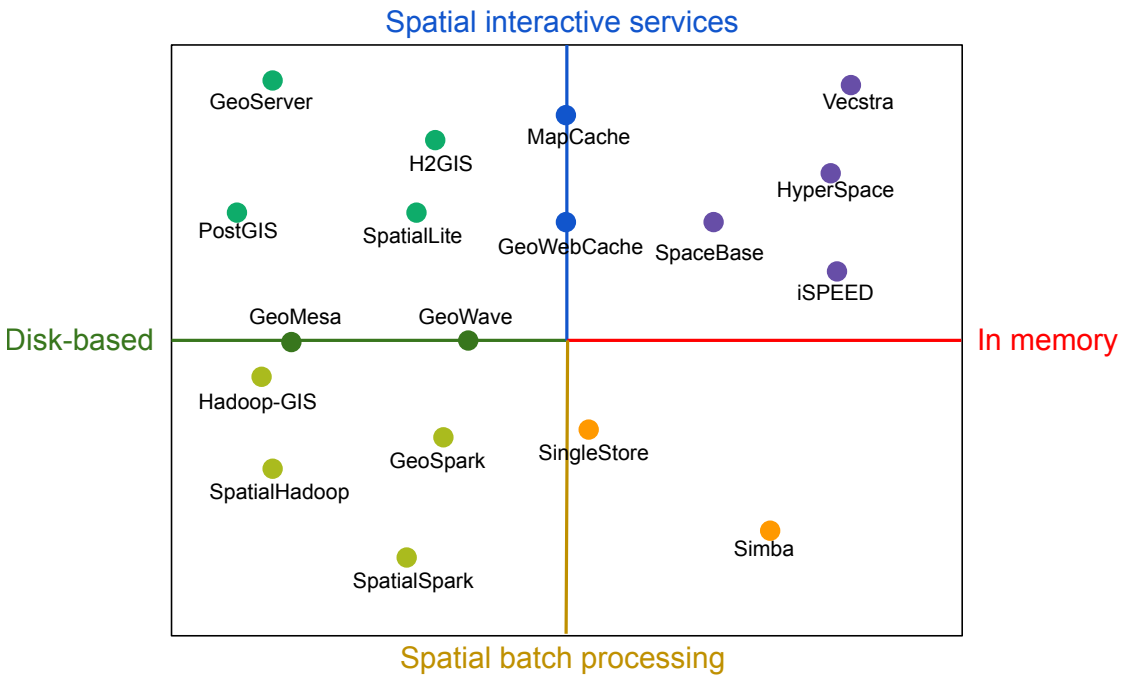


Figure 1.1: Spatial Data Management Systems Summary.

categorize into two dimensions: spatial interactive services vs. spatial batch processing; and disk-based vs. in-memory, as shown in Figure 1.1.

There are disk storage databases for batch processing and multiple types of spatial queries such as SpatialHadoop (Eldawy and Mokbel, 2015; Eldawy and Mokbel, 2013), Hadoop-GIS (Aji *et al.*, 2013), GeoSpark (Yu *et al.*, 2015; Yu *et al.*, 2016) and SpatialSpark (You *et al.*, 2015). Disk storage databases can be used as repositories for analysis of historical mobile IoT data, but they are not adequate for services that require online interaction with IoT data.

Other than batch processing and analysis, a number of specialized systems have been developed to support online interactive requests, often through short database transactions. For instance, PostGIS (Holl and Plum, 2009) is a geospatial extension to PostgreSQL (Momjian, 2001) that supports spatial SQL queries. GeoServer (Deoliveira, 2008) is built on PostGIS and designed for interoperability. It implements industry-standard OGC protocols (OGC, 2021). H2GIS (Bocher *et al.*, 2015) to H2 (*H2 Database Engine* 2021) and SpatialLite (Furieri, 2014) to SQLite (Owens and Allen, 2010) are both as PostGIS is to PostgreSQL. GeoMesa (Hughes *et al.*, 2015) integrates OGC APIs and protocols as well, and it also supports Apache Spark for distributed geospatial analytics. GeoWave (Annex, 2018) is a similar spatial library that can provide OGC services and Map-Reduce processing for distributed analysis of geospatial data.

Despite the popularity of spatial databases such as PostGIS, their design has not benefited substantially from developments in in-memory databases. PostGIS still utilizes disk as its primary storage, which raises the problem to reduce expensive geospatial data querying cost due to disk I/O and system overheads. Since IoT applications are experiencing growing popularity, this problem is made worse by large numbers of concurrent requests. With the increasing capacity and the decreasing price of memory, there is a greater possibility to cache substantial amounts of spatial data in memory. Storing data in memory is especially valuable for running concurrent workloads as it eliminates the inevitable bottlenecks caused by disk-centric architectures (Stonebraker *et al.*, 2007), while at the same time increasing throughput and resource utilization.

Disk-based spatial databases utilize main-memory by adding caches, such as GeoWebCache (Nie *et al.*, 2011) integrated with GeoServer, or MapCache (Bonfort and Bonfort, 2013), which is a disk cache similar to GeoWebCache. However, disk-based caches cannot eliminate the I/O cost and system overheads, while fully in-memory spatial data platforms can achieve that by design.

To the best of our knowledge, only a few data management platforms stand out in this category. One is HyperSpace (Pandey *et al.*, 2016), a geospatial main-memory database system that can efficiently process geospatial queries. The other one is SpaceBase (Universe, 2012), a system that handles concurrent spatial queries with low latency based on R-Tree indexing. iSPEED (Vo *et al.*, 2018) is an in-memory spatial query system for large and structurally complex 3D data. Vecstra (Wang, 2018) is an efficient and scalable OGC standards-compliant in-memory geospatial cache. Simba (Xie *et al.*, 2016) is a spatial in-memory system for big data analytics. SingleStore (*SingleStore: All Data, One Platform*. 2021) is designed to fit all data within the main memory and has recently been extended to support disk-based OLAP workloads.

These in-memory spatial databases can efficiently support queries and updates of spatial data. However, these described in-memory solutions have limited support for reactivity features that are required in reactive mobile IoT applications.

1.2.2 Actor-Oriented Data Management Systems

As reactive IoT applications demand scalability and elasticity, as well as efficiency in distributed resource utilization, actor programming models have attracted attention in this setting (Haller, 2012; Hewitt, 2010).

Actors comprise a programming model that provides high concurrency and distribution (Agha, 1990). The inherently distributed property of actors make them ideal building blocks to provide a scalable computing infrastructure when building IoT applications.

Actors provide modularity by keeping private states and only modifying other actors' states via immutable asynchronous messages (Bowers and Ludäscher, 2005). Actors encapsulate different logic and tasks, which is a perfect match for heterogeneous IoT devices with distinct functionality. An encapsulated state in an actor is an alternative to conventional shared-memory concurrency, which eliminates the need for the developer to ponder about complex lock-based synchronization, thus being a huge help for frequently changed IoT data.

There are popular actor programming implementation options such as Akka (*Akka - Build Powerful Reactive, Concurrent, and Distributed Applications More Easily* 2020), Erlang (*Erlang-Build massively scalable soft real-time systems* 2020), Orleans (Bykov *et al.*, 2011; Bernstein *et al.*, 2014), and Orbit (*Orbit* 2020). In particular, Orleans provides a virtual actor abstraction that treats actors as entities in perpetual existence. Deactivation and activation of virtual actors are automatically taken care of by the Orleans runtime. Virtual actors can help ease the burden on developers by providing support for fault tolerance and resource management (Bernstein and Bykov, 2016). Moreover, Orleans is inherently scalable and distributed, and used in many Microsoft research projects and products (*Halo: Combat Evolved* 2021; Sarwat *et al.*, 2012). Orleans is an open-source project implemented in C#, while Orbit is a Java implementation inspired by the virtual actor abstraction.

Despite the great advantages of the actor model and its potential fit to IoT applications, this model of computation has not been sufficiently explored in the design of IoT data platforms. An Actor-Oriented Database (AODB) (Bernstein, 2018) is a data management solution that integrates classic DBMS features with the actor programming model so as to provide database properties, such as transactions (Bykov *et al.*, 2011) and indexing (Bernstein *et al.*, 2017b).

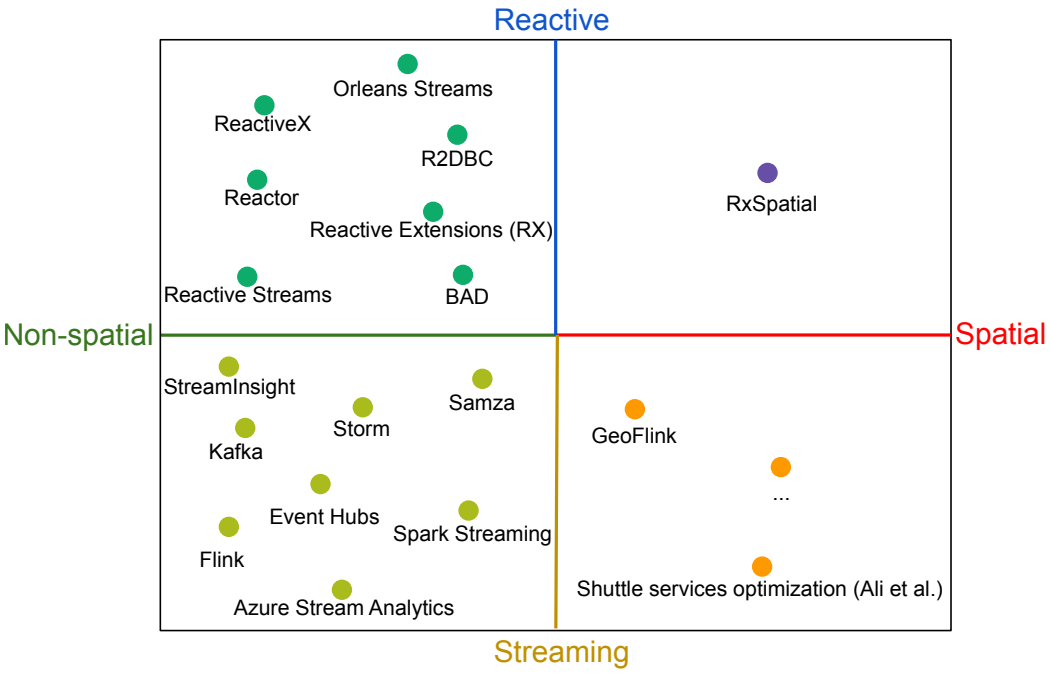


Figure 1.2: Reactive Data Management Systems Summary.

AODBs are an attractive choice for building IoT data platforms. However, current AODBs lack spatial data and reactive functionality support.

1.2.3 Reactive Data Management Systems

Reactivity is one crucial feature in many IoT applications. IoT entities continuously move and trigger reactions based on nearby events and execute queries to explore their surroundings. A reactive mobile IoT data platform needs to support spatial reactivity functionality. A wide range of solutions in commercial systems and previous research efforts has explored how to provide data-driven operations and reactivity. To meet IoT data platforms' requirements, we categorize them into two dimensions to ease discussion: Reactive vs. Streaming, Spatial vs. Non-spatial, shown in Figure 1.2.

Kafka (*Apache Kafka* 2021), Flink (Carbone *et al.*, 2015), Samza (*Apache Samza - A distributed stream processing framework* 2021), Event Hubs (*Event Hubs - Simple, secure, and scalable real-time data ingestion* 2021), Storm (*Apache Storm* 2021), Microsoft StreamInsight (*Microsoft StreamInsight* 2020) and Spark Streaming (*Apache Spark Streaming* 2021), to name a few, are popular stream processing and analysis frameworks. There are also some solutions that provide spatial event monitoring using streaming systems. For instance, GeoFlink (Shaikh *et al.*, 2020) extended Flink to support spatial data types,

indexes, and continuous queries over spatial data streams. The studies conducted in (Ali *et al.*, 2010a; Ali *et al.*, 2010b) combine Microsoft SQL Server Spatial Library (*Microsoft SQL Server Spatial* 2020) with StreamInsight to build a demo for shuttle service optimization by using spatial continuous queries and various real-time analytics. Works in (Kazemitabar *et al.*, 2010; Miller *et al.*, 2011) support continuous spatial queries for monitoring as well. (Galic *et al.*, 2017) further proposed a distributed spatio-temporal mobility data stream framework to support continuous queries over streams.

The streaming solutions we mentioned above often expect a topology of stream operators, which are suitable for specifying a set of bulk operations over a large number of data items with fairly regular schemas, such as transformation, filtering, join, and aggregate. However, it is unnatural and difficult to use these solutions to specify complex application logic, such as heterogeneous operations over varied data items as necessary in reactive IoT applications. Meanwhile, the described streaming systems have limited support for external functions that may be triggered in the reactive operations. Luckily, there are some reactive programming frameworks that target solving those scenarios.

Reactive systems are driven by messages and require responsiveness in a timely manner, in the face of failure, and under varying workloads as stated in the Reactive Manifesto (Bonér *et al.*, 2014). To be noticed, reactive is not the opposite of streaming. We see reactive behavior as a counterpart to event streams in interactive data management scenarios. Reactive enriches event streams with interactive data processing logic, i.e., responding when some underlying data changes.

Reactive Relational Database Connectivity (R2DBC) (*The Reactive Relational Database Connectivity (R2DBC)* 2021) provides a reactive API that can integrate with common SQL databases such as Microsoft SQL Server, PostgreSQL, and MySQL. Orleans Streams (*Orleans Streams - Microsoft Orleans Documentation* 2020) is a streaming extension that helps achieve reactivity on top of Orleans. It allows developers to write reactive functions that operate over events in a structured way. ReactiveX (*Microsoft Reactive Framework - ReactiveX: An API for asynchronous programming with observable streams* 2020) supplies observable streams that provide reactivity in asynchronous programming. Reactive Streams (*Reactive Streams* 2021) is designed on the JVM for reactivity based on asynchronous stream processing. Reactor (*Reactor - Create Efficient Reactive Systems* 2021) is based on the Reactive Streams for helping build

non-blocking applications. Reactive Extensions (RX) (*Reactive Extension* 2020) serve reactions to asynchronous events based on an observable interface. A Big Active Data (BAD) system is introduced by (Jacobs *et al.*, 2020) and combines Apache AsterixDB with an active toolkit, which can publish information to subscribed users in a timely fashion. However, spatial data management is not taken into consideration in these reactive solutions.

To the best of our knowledge, one of the most related reactive spatial data management solution to our work is RxSpatial (Shi *et al.*, 2016a). RxSpatial provides real-time spatial-aware reactivity by combining Microsoft SQL Server Spatial Library and ReactiveX (Hendawi *et al.*, 2016). However, RxSpatial only monitors the relation between explicitly subscribed objects, which is not a practical solution in dynamically changing IoT applications, making it hard to maintain performance for varied workloads.

Summary. In-memory spatial databases efficiently support data processing by reducing the I/O cost and overheads. However, they fail to fully satisfy the characteristics necessary for building reactive mobile IoT data platforms, such as support for heterogeneous data representations, easy of management towards scalability and elasticity, and support for reactivity functionality. Distributed actor systems can help solve some of these problems by providing inherent modularity and scalability. However, actor systems lack support for reactivity and spatial data management. Current reactive systems have limited support for defining flexible operations over heterogeneous data items and support spatial features in a limited and constrained way.

To provide a scalable and reactive data management for mobile IoT applications, we aim to explore how to utilize the actor programming model to build IoT data platforms. In addition, we investigate how to build a scalable mobile IoT data management system that integrates reactivity functionality.

1.3 Summary of Goals and Contributions

The main objective of the work presented in this dissertation is to explore abstractions and systems for developing scalable and reactive data management for mobile IoT applications. To achieve this goal, firstly, we illustrate new and distinct requirements of building IoT data platforms, elaborate on the challenges of modeling IoT data platforms, discuss why actor-oriented databases are naturally suitable for satisfying these requirements and solving

these challenges, then provide guidelines for modeling and building IoT data platforms with AODBs. Afterward, to ease the burden of developers and to reduce the complexity of building reactive mobile IoT data platforms, we propose a novel abstraction for reactive moving objects and utilize the proposed model of moving actors to discuss the architecture of a new data management system called Moving Actor-Oriented Databases (M-AODBs). A first implementation, Dolphin, is presented to validate the design of M-AODBs. In the end, to handle the typically skewed spatial distributions in IoT applications, we apply varied classic partitioning techniques in Dolphin to evaluate, analyze, and manage bottlenecks due to data skew.

1.3.1 Modeling and Building IoT Data Platforms with Actor-Oriented Databases

Due to the unprecedented development of IoT applications, the massive amount of data generated from IoT applications has brought about new challenges associated with data processing and management.For instance, it is needed to support data variety, efficient processing under high degrees of concurrency, as well as scalability and elasticity. Those challenges necessitate the exploration of new ways to build IoT data management and processing platforms.

In this part of the thesis, we investigate the functional and non-functional requirements for IoT data platforms through two case studies. One is Structural Health Monitoring (SHM) systems that utilize sensors to identify damaged sections on parts of large constructions that can cause safety concerns. The other one is Beef Cattle Tracking and Tracing systems that refer to a part of the beef cattle supply chain. The IoT application utilizes sensors to provide cow tracking information and help retailers and consumers trace meat products. Based on the requirements, we show the challenges to be met in constructing IoT data platforms. We advocate that Actor-Oriented Databases (AODBs) are suitable for solving these challenges by providing a modular, stateful, and scalable substrate. Then, we illustrate how to model the case studies with AODBs and provide guidelines for how to solve typical questions in the actor modeling process effectively. Following the proposed guidelines, we implement an SHM data platform based on AODBs.

1.3.2 Dolphin: An Actor-Oriented Database for Reactive Moving Object Data Management

Reactivity has been seen as a feature that gets increasing attention in varied mobile IoT applications scenarios. Current reactive mobile IoT data platforms enrich well-researched spatio-temporal data management with reactivity logic in the application tier, which is undesirable due to the resulting high complexity of application development and the overheads of data shipping, making it challenging to meet the tight low-latency constraints of reactive functionality. Placing application logic together with the data while providing correctness under concurrency are desiderata to facilitate the construction of complex mobile IoT applications.

In this part of the thesis, we present an actor-based abstraction for reactive moving objects, moving actors, to represent them in an Actor-Oriented Database. Then, we explore Moving Actor-Oriented Databases (M-AODBs) – a new system that integrates reactive functionality into a distributed actor framework with spatial data management. Based on an analysis of reactive moving object application use scenarios, two data concurrency semantics are presented to outfit this distributed asynchronous system. One is Actor-Based Freshness semantics, which is used in scenarios needing real-time reactions. The other one is Actor-Based Snapshot semantics, which can provide consistent images of the locations of moving actors. After we present the design and architecture of M-AODBs, a first implementation of M-AODBs called Dolphin is discussed. Our experiments show that Dolphin can achieve the requirements of scalability and low-latency reactions for reactive moving object scenarios.

1.3.3 An Evaluation of Spatial Partitioning Techniques to Handle Skew in Moving Actor-Oriented Databases

M-AODBs utilize spatial partitioning to achieve scalability through parallelism and distribution. Through spatial partitioning, queries, moves, and reactions in M-AODBs are processed across logical partitions based on their spatial attributes. Therefore, spatial partitioning influences the load balance in M-AODBs. Besides, spatial skew is a common situation in reactive moving object applications. Skewed data can make load balance hard to achieve in M-AODBs,

which may not only drag down the performance of M-AODBs but also increase the long-tail latency of reactions. However, the impact of different partitioning methods in M-AODBs under spatial skew is unknown.

In this part of the thesis, we evaluate varied partitioning techniques in M-AODBs under skewed spatial data. We aim to find the critical factors in M-AODBs for keeping consistent performance and providing low-latency reactions. First, we illustrate how spatial partitioning is integrated in M-AODBs. We show how to adapt several classic partitioning techniques to the concrete M-AODB Dolphin and conduct experiments to show their impact. Then, we give guidance on how to choose partitioning methods in different scenarios and point out aspects that can be further optimized in the implementation of M-AODBs.

1.3.4 Publications

Previous versions of some of the chapters of this dissertation have been published or are in submission or will be submitted to international conferences and journals. We list these chapters along with their respective publications below:

- **Chapter 1 - Vecstra: An Efficient and Scalable Geo-spatial In-Memory Cache (Wang, 2018).** Yiwen Wang. Proceedings of the VLDB 2018 PhD Workshop co-located with the 44th International Conference on Very Large Databases (VLDB 2018). Rio de Janeiro, Brazil, Aug 27-31, 2018.

- **Chapter 2 - Modeling and Building IoT Data Platforms with Actor-Oriented Databases (Wang *et al.*, 2019b)**[1] Yiwen Wang, Julio Cesar Dos Reis, Kasper Myrtue Borggren, Marcos Antonio Vaz Salles, Claudia Bauzer Medeiros, Yongluan Zhou. Advances in Database Technology — EDBT 2019 Series ISSN: 2367-2005. Proceedings of the 22nd International Conference on Extending Database Technology. Lisbon, Portugal, March 26–29, 2019. page 512-523. DOI: 10.5441/002/edbt.2019.47.

[1]The paper contains data material that has also formed part of a MSc thesis - Scalable Structural Health Monitoring Data Platform using Actors as a Database, by Kasper Myrtue Borggren. All work about this paper (i.e., designing of the work; conducting research; interpreting of data; drafting and revising the manuscript) have been produced as part of the PhD study except experiment coding and data generation.

- **Chapter 3 - Dolphin: An Actor-Oriented Database for Reactive Moving Object Data Management.** Yiwen Wang, Vivek Shah, Marcos Antonio Vaz Salles, Claudia Bauzer Medeiros, Julio Cesar Dos Reis, Yongluan Zhou. Manuscript in submission. April, 2021.
- **Chapter 4 - An Evaluation of Spatial Partitioning Techniques to Handle Skew in Moving Actor-Oriented Databases.** Yiwen Wang and Marcos Antonio Vaz Salles. Manuscript in preparation. April, 2021.

1.3.5 Structure of Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we explore the problem of modeling IoT data platforms based on two case studies and explain why we choose a recently proposed approach, Actor-Oriented Databases (AODBs), to build IoT data platforms. We then provide guidelines on how to model and build IoT data platforms using AODBs and verify their applicability in an implementation of one of the two case studies. In Chapter 3, we focus on the construction of IoT data platforms for reactive moving object applications. We introduce a new abstraction of moving actors for reactive moving objects, then present a new reactive moving actor data management platform – namely Moving Actor-Oriented Databases (M-AODBs). We also provide a realization of M-AODBs named Dolphin. Our experiments on Dolphin validate our proposal of M-AODBs and allow us to observe its scalability and reactivity characteristics. In Chapter 4, we extend Dolphin with a variety of spatial partitioning techniques to improve the throughput and reduce tail latencies of Dolphin in the presence of spatial skew of reactive moving object applications. In Chapter 5, we summarize the dissertation and outline future research directions.

2 Modeling and Building IoT Data Platforms with Actor-Oriented Databases

Vast amounts of data are being generated daily with the adoption of Internet-of-Things (IoT) solutions in an ever-increasing number of application domains. There are problems associated with all stages of the life cycle of these data (e.g., capture, curation and preservation). Moreover, the volume, variety, dynamicity and ubiquity of IoT data present additional challenges to their usability, prompting the need for constructing scalable data-intensive IoT data management and processing platforms. This chapter presents a novel approach to model and build IoT data platforms based on the characteristics of an Actor-Oriented Database (AODB). We take advantage of two complementary case studies – in structural health monitoring and beef cattle tracking and tracing – to describe novel software requirements introduced by IoT data processing. Our investigation illustrates the challenges and benefits provided by AODB to meet these requirements in terms of modeling and IoT-based systems implementation. Obtained results reveal the advantages of using AODB in IoT scenarios and lead to principles on how to effectively use an actor model to design and implement IoT data platforms.

2.1 Introduction

Internet-of-Things (IoT) systems enable data interactions through machine-to-machine communication stemming from supporting devices connected to the Internet (Bandyopadhyay and Sen, 2011). IoT systems generate a potentially huge amount of data from devices that dynamically enter and leave the IoT environment, with very high-speed data flow and processing. Data, in turn, are generated by a wide variety of devices, thus giving rise to highly heterogeneous data streams. In this work, we distinguish between *IoT systems* (i.e., the entire software ecosystem involved in an IoT scenario) and *IoT data platforms* (i.e., the data and data management software modules that are part of an IoT system). Our work focuses on the latter.

Enormous challenges need to be addressed in order to realize the full potential of IoT. First, there is a tension between effective data management and fulfillment of performance requirements in IoT data platforms. Indeed, many IoT systems are processor-intensive and require processing a massive amount of highly concurrently generated data. The management of these interactions among data with low latency remains an open research problem. Second, being able to deal with dynamic scaling while guaranteeing protection of data from different entities is another significant challenge. Therefore, we focus our investigation on how to manage the data from large volumes of devices and, at the same time, ensure the dynamic and flexible development of applications. This dual aim must be achieved while respecting application constraints for low latency in interactive functionality as well as data protection and access control.

Given these characteristics, we propose that actor-oriented databases (AODBs) are ideally suited to manage the data of real-world IoT systems. Actors comprise a model of computation specifically aimed at high concurrency and distribution (Agha, 1990). To that effect, actors keep their private states and can modify states by communicating with each other via immutable asynchronous messages (Bowers and Ludäscher, 2005). As such, actors are natively applicable to support the management of an arbitrary number of independent and heterogeneous streaming data sources. AODBs, in turn, enrich actors with classic RDBMS functionality by integrating data management features, such as indexing, transactions, and query interfaces, into actor runtimes (Bernstein *et al.*, 2017b). These features make AODBs attractive for building an IoT data platform. In more detail, AODBs stand out for several reasons. First, IoT systems comprise many different devices with distinct functionality. This requirement is directly met by the actor model, through the principle of assigning different logic and tasks to actors. Second, in IoT, data changes frequently; actors provide a natural alternative to conventional concurrency models that rely on synchronization of shared mutable state using locks. Third, the characteristics of non-blocking interactions via immutable messages between actors match well with the demands of IoT systems. Fourth, the number of actors can scale out quickly without consuming excessive resources. Dynamic scaling is a common situation in IoT in which all kinds of sensing devices (including humans!) can quickly enter – but also leave – a system.

There are several examples of the use of actors in IoT scenarios (*Akka Documentation, Version 2.5.17, IoT example use case* 2018; Persson and Angelsmark,

2015; Sánchez *et al.*, 2015; *Who Is Using Orleans?* 2018). However, these previous studies concentrate on implementation aspects, neither providing guidance on how to model IoT data platforms with actors nor analyzing the fit of AODB to the requirements and challenges brought about by IoT. By contrast, to the best of our knowledge, this work is the first that builds an end-to-end case for the suitability of AODBs to manage IoT data, going from requirements and modeling to implementation and performance evaluation. Our work covers a wide gamut of issues to justify and showcase the adoption of AODBs as an appropriate solution to meet the main challenges of data management in IoT systems. Our main contributions are therefore:

1. We discuss core requirements of IoT data platforms, and challenges to be met in their implementation. We illustrate this discussion through the analysis of two real world IoT case studies.

2. We present a methodology and guidelines to model an AODB for such platforms.

3. We develop a prototype of one of the case studies and present its evaluation to show the effectiveness of adopting AODBs for IoT data platforms.

The remaining of this chapter is organized as follows. Section 2.2 presents two case studies of IoT systems, which we use to extract functional and non-functional requirements, and to present some of the major challenges to be faced. Section 2.3 justifies our choice of AODBs as an appropriate technology to meet such requirements and challenges. In Section 2.4, we provide a detailed discussion of the challenges of modeling such platforms, and show how these challenges can be overcome for the running cases. Sections 2.5 and 2.6 respectively present our prototype for one of these cases and its evaluation. Section 2.7 revisits the work by contrasting it with related work. Finally, Section 2.8 presents conclusions and ongoing work.

2.2 IoT Data Platform Case Studies

In this section, we discuss the requirements for an IoT data platform and analyze two specific case studies. There are several scenarios in IoT data platforms, such as healthcare, personal security, traffic control, environmental monitoring, and disaster response. The two IoT data platform cases that we focus on are drawn from our experience with a structural health monitoring

system (cf. Subsection 2.2.1) and a beef cattle tracking and tracing system (cf. Subsection 2.2.2). We have worked directly with these case studies, helping us validate common non-functional requirements for IoT data platforms as well as collect illustrative functional requirements for these applications.

For the first study, we have cooperated with SenMoS (*SenMoS: your sensor monitoring system* 2018) in the area of structural health monitoring for large constructions, e.g., bridges. SenMoS is a Danish company that provides users with entire monitoring solutions, including requirement elicitation and cloud data management. The developers at SenMoS have participated in the design and implementation of the IoT data platform for the Great Belt Bridge (*Facts and History* 2018). The second case focuses on the management of cattle produce (and in particular the beef supply chain) from the perspective of traceability. This study is based on previous work with domain experts from the Brazilian agricultural research corporation Embrapa (*Brazilian Agricultural Research Corporation - A Embrapa* 2018) that studied traceability in food for supply chains (Kondo *et al.*, 2007), and on interactions within the Danish Future Cropping partnership (*Future Cropping partnership website* 2018), particularly with experts from the agriculture solution provider SEGES (*SEGES Landbrug & Fødevarer F.m.b.A. website* 2018). Both of these organizations have substantial experience in the agricultural sector and are key players in agricultural extension systems of the respective countries. Both case studies concern the development of a scalable data platform that collects and stores data from IoT devices, processes operations, and provides information services to different users. Although these two IoT platforms target different scenarios, they present several common aspects and non-functional requirements. In particular, the systems should operate as Software-as-a-Service (SaaS) solutions and thus manage the data from several different tenants. Moreover, it is desired that scalability to large data volumes or users be achieved without a high burden on the data platform developers.

Non-Functional Requirements for IoT Data Platforms

We elicited the following common non-functional requirements shared by different IoT data platforms:

1. **Data ingestion from endpoints**. The IoT data platform must have the capability to receive and store data from IoT devices, e.g., GPS collars on cows.

2. **Multi-tenancy**. The IoT data platform must provide varied information services to different users.

3. **Support for heterogeneous data**. The IoT data platform must be modular in its support for data ingested from IoT devices and allow for communication employing different data formats.

4. **Cloud-based deployment**. The IoT data platform can be distributed in the cloud for ease of operation, management, and maintenance.

5. **Scalable data platform**. The IoT data platform must not degrade in functionality or performance while expanding. This must occur without modifying existing software components.

6. **High efficiency**. The IoT data platform must process massive amounts of concurrently generated data effectively.

7. **Access control and data protection**. The IoT data platform should support data protection, enforcing authentication and access control over different users and profiles.

In addition to the requirements above, it is often the case that IoT data platforms must serve queries over historical data accumulated from devices over long time periods. In this chapter, we focus, however, on online data ingestion and querying in SaaS scenarios. We note that at present, there is only limited support for declarative multi-actor querying in AODBs (Bernstein *et al.*, 2017b), and thus complex historical analyses could still be served by a data warehouse.

2.2.1 Case Study 1: Structural Health Monitoring

Structural Health Monitoring (SHM) systems aim to identify damaged sections on parts of large constructions that can cause safety concerns. SHM systems can help organizations save time on inspections by gathering and processing data so that the system can generate alerts when problems arise or suggest actions that can prevent faults. SHM systems are equipped with a set of sensors, e.g., to measure a bridge's extension, inclination, temperature, wind speed, and wind direction. Each sensor is connected to a data logger that converts the sensors analog signal into a digital one. The platform must collect, process and store data from the sensors. Figure 2.1 presents a context diagram of the Structural Health Monitoring Data Platform. This design is based on a real case

study. Sensors provide data to different stakeholders, e.g., engineering experts monitoring the structure, data analysts, or the maintenance personnel who manages the monitoring projects. The SHM system must meet the following functional requirements:

1. The system must control several construction structures (e.g., bridges) using the same data platform.

2. The system must be structured to support data storage, i.e., data must be saved in a way that allows for further data manipulation and analysis.

3. The data platform must be able to maintain data from multiple sensors, users, projects, and organizations.

4. The data platform must calculate the accumulated change for each data stream from a sensor, e.g., to gauge how far elements have moved when using extension sensors.

5. The data platform must send customized alerts to users when thresholds are met, depending on individual sensors or sensor types. Thresholds can be used for determining the need for maintenance, or to call attention to ongoing events.

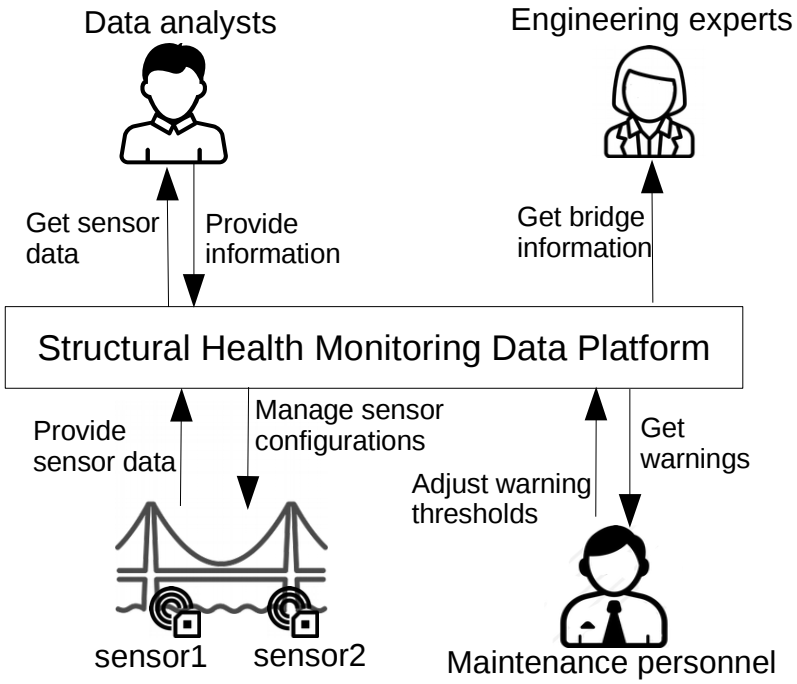


Figure 2.1: Context Diagram for Structural Health Monitoring Case Study.

6. The data platform must support plots providing statistical aggregates to help users spot meaningful events in time series. Besides, online plotting of recent raw sensor data is required to let personnel explore events interactively.

7. The data platform must allow for browsing of live data from sensors, along with continuously derived equations, to provide a view of the current state of the structure.

2.2.2 Case Study 2: Beef Cattle Tracking and Tracing

Agricultural supply chains involve a complex network of producers, retailers, distributors, transporters, storage facilities, and consumers in the sale, delivery, and production of a particular product. Trackability and traceability are essential requirements in food marketing (Cimino *et al.*, 2005). Tracking refers to following the path of an entity from the source to destination. Tracing refers to identifying original information regarding an entity and tracing it back in the system (Mousavi *et al.*, 2002). Systems for tracking and tracing agricultural products increase consumer confidence on provenance and quality of the food they buy, while at the same time helping retailers and certification authorities to monitor products.

The ability of IoT to collect data from sensors as well as trace entities is a crucial enabler for monitoring such chains. Systems that automate tracking and tracing in an agricultural supply chain should not only collect data, but also connect users and objects at any place and time. Data integration, processing, analysis and service support present many challenges in this context. For the sake of feasibility, we assume for this case study, similarly to other food tracing systems (*IBM Food Trust: trust and transparency in our food* 2018), that a global standard for supply chain messages, GS1 (*Global Standards One* 2018), is adopted by participants that connect with the IoT data platform. As such, we do not discuss the data integration problem in this chapter.

Our case study refers to a part of the beef cattle supply chain, concentrating on cow tracking and meat product tracing, providing tracking information and helping consumers trace meat products. Figure 2.2 presents the entities interacting within the data platform. The system must provide multi-tenancy services to host the data of different participants and supply chains. From a

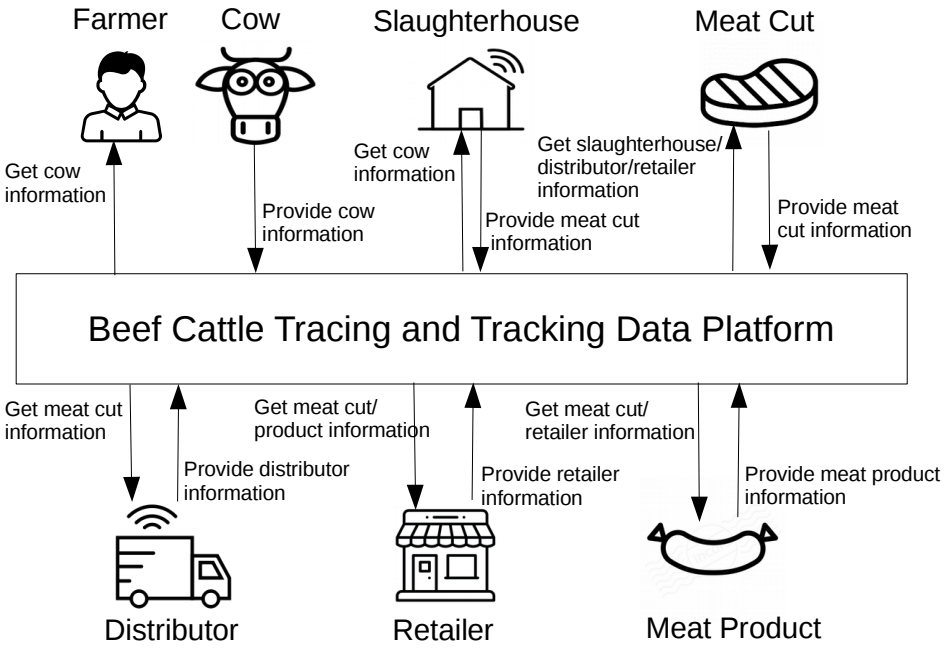


Figure 2.2: Context Diagram for Beef Cattle Tracking and Tracing Case Study.

high-level point of view, there are five kinds of tenants in our system: farmers, slaughterhouses, distributors, retailers, and consumers. Each involved part is the source of different types of data in the system to enable the tracing of the whole life-cycle of a given meat product.

Full-fledged cattle sensor-based systems involve the deployment of very many kinds of sensors – both in individual animals and in their environment. For instance, each animal has external sensors (e.g., collars, earrings) to measure movement, speed, location. Cattle often also have sensors inside their digestive tract (usually swallowed, sometimes implanted), to measure factors such as temperature, metabolic variables, or digestive characteristics. Environmental sensors may monitor factors such as cattle weight, or soil humidity. Additional sensors along the supply chain include devices that provide trackability (e.g., in transportation), but also traceability and quality (e.g., monitoring temperature inside warehouses). Without loss of generality, we have simplified this scenario to consider only a few of these sensing sources, keeping only enough distinct sensors to illustrate actual data and sampling rate heterogeneity. This simplified scenario must fulfill the following functional requirements:

1. The data platform must store the data from animal and environment sensors, such as collars bound to each individual cow, to enable retrieval of location, motion, and other facts regarding traceable entities.

2. Farmers need to track each cow's trajectory and behavior, and thus the data platform must record the locations of each cow over time. Geo-fencing can help identify whether a cow is in an appropriate area (e.g., when rotating pasture grounds) (Breen, 2009).

3. Slaughterhouses wish to access services that provide information about cows that will be slaughtered. For instance, it must be possible to access tracing information such as the provenance of the cows and tracking information about where the meat cuts produced after slaughter are transferred to.

4. Distributors wish to get tracing information of a meat cut and tracking information of where those cuts are to be sent to.

5. Retailers aim to know the source of the meat cuts and manage their transformation into meat products for consumers.

6. Consumers wish to get tracing information about meat products over the whole supply chain.

2.2.3 Challenges for IoT Data Platforms

The functional and non-functional requirements discussed in this section render the modeling and building of IoT data platforms a non-trivial undertaking. The construction of such a platform involves technical issues related to capturing, identifying and storing relevant events, managing associated constraints, processing varied types of queries, etc. Further complexity arises from taking into account the necessities of different stakeholders, and issues related to data precision, synchronization and availability.

Choosing the right database architecture is therefore a key decision for the success of an IoT data platform project. Virtualized deployment for efficiency and ease of scaling to large request volumes are significant obstacles (Shah and Salles, 2018a). Moreover, support for multi-tenancy needs to be carefully designed. While physical sharing of tenant data lowers overhead and increases efficiency (Aulbach *et al.*, 2008), this strategy opens up security risks, which are related to the lack of modularity at the database level (Shah and Salles, 2018b).

Thus, several questions need to be addressed when building IoT data platforms. First, a vast amount of data is concurrently generated from IoT devices. How

can this data be managed and processed in the data platform? Second, how can data protection and access control across different entities be enforced while sharing data effectively? Third, our case studies suggest that a variety of queries, including analyses of time series from bridge sensors, spatial queries for cow locations, or graph navigation for tracing, need to be efficiently supported over IoT data. How can applications be modeled and built to support different types of queries? Fourth, it is necessary that the system be easily scaled without affecting functionality and performance. How can the platform be architected to easily scale out when it becomes necessary to manage more users and data? In this investigation, we observe that the issues regarding modularity and scalability pointed out in this section can be simultaneously addressed by AODBs (Bernstein *et al.*, 2017b). We design cloud-centric actor-oriented database backends for the two IoT case studies introduced in this section. As a first step, we explain the motivation of taking an approach based on AODBs in the next section.

2.3 Why Actor-Oriented Databases?

We argue that an actor-oriented database is the ideal organization for an IoT data platform, enabling fulfillment of all common non-functional requirements identified in Section 2.2. Moreover, AODBs ease the achievement of functional requirements by providing a modular, stateful, and scalable substrate for the modeling, design and implementation of an IoT data platform. The following characteristics of an AODB illustrate its suitability to address the challenges of IoT data platforms.

AODBs facilitate the management of distribution and the encapsulation of data.

Actors are logically distributed, and can thus naturally map to dispersed entities such as sensors. The latter promotes the expression of parallelism in the application logic responsible for data ingestion into the platform. Moreover, an AODB-centric design functionally decomposes the data platform into different actors. State is encapsulated within each actor, and can only be communicated by asynchronous messages. As such, actors provide a mechanism for isolation of different functions and data, enabling efficient support for multi-tenancy.

Actor modularity in AODBs supports representation and sharing of heterogeneous data.

Actors are the unit of modularity in an AODB. By encapsulating state and supporting specification of user-defined APIs, actors abstract heterogeneous data representations. Moreover, arbitrary data transformations can be coded in actor methods, enabling asynchronous exchange of data across heterogeneous actors. As such, actors offer an attractive model to capture heterogeneity of data formats and representations originating at multiple IoT devices.

AODBs employ multiple actor types and concurrent execution among actors to achieve scalability.

The support for multiple actor types enables the representation of different kinds of entities in the IoT data platform. When a new entity type is added to the system, it is represented by a new actor type added to the data platform. AODBs thus support a gradual extension of the platform through new actors and actor types with minimal impact on existing components. The use of actors makes scaling out easier, since new actors can be deployed over additional hardware components to avoid violation of performance constraints. The resulting concurrent and distributed execution facilitates efficient use of computational resources to bolster scalability.

Parallelism across actors in AODBs allows for processing of massive amounts of concurrently generated data.

By identifying tasks and associated logic among entities, we can model entities as independent actors, so that they can perform tasks concurrently. When independent tasks are then run in parallel across actors deployed on separate hardware components, data platform performance can be improved. Since sensors are naturally modeled as different actors, parallel execution can be leveraged in processing data from a large number of data streams.

Encapsulation and modularity in AODBs support data protection and access control.

As data in one actor are invisible to others, access permissions can be checked when data are exchanged by asynchronous messaging (*Grain Call Filters* 2018). In other words, data are protected inside an actor, and mechanisms for access control can ensure data are only shared with authorized users.

2.4 Modeling the Case Studies with Actor-Oriented Databases

AODBs provide scalable data processing, management, and storage over a set of application-defined actors (Bernstein *et al.*, 2017b). However, to the best of our knowledge, there is scant guidance on how to model applications to reap the benefits of AODBs. In this section, we fill this gap by an in-depth modeling discussion of the two IoT case studies described in Section 2.2. We contribute to the construction of IoT data platforms in two ways: (1) we provide guidelines for modeling IoT data platforms with AODBs; and (2) we explain how an actor model affects the system both in design and implementation.

It is believed that application modeling helps to preserve and reuse information in other projects, as well as facilitates the automated generation of a system from models (Teorey, 1999). To support the latter aim, we leverage UML notation (*OMG Unified Modeling Language (OMG UML) Version 2.5.1* 2018) to create models of actors, their encapsulated state and their operations. These data-centric models harken back to conceptual modeling approaches in databases, enabling both specifications of data requirements and, in the future, code generation for AODB platforms. To support the former aim, we focus on documenting database actors and their asynchronous interactions. In addition to database actors, the classic architecture of an IoT data platform contains a stateless tier that mediates the interaction with users or devices. The analysis of this tier is outside the scope of our work; we abstract its functionality as stateless actors operating as proxies and omit this tier from our models.

In the presented models, we represent the minimal necessary information to emphasize techniques for actor-oriented database design. Application details that would make the presentation unnecessarily complex are thus omitted, and simplifications are made where appropriate. In the sections that follow, we identify each core modeling question encountered in the two IoT data platform case studies and discuss lessons learned.

2.4.1 How can Actors be Identified?

A variety of entities exist in any system, and these entities either perform or collaborate to achieve different tasks. Moreover, these entities have different life cycles, distinct types, and varied needs regarding heavy computation or

communication (Bernstein, 2018). To take advantage of the actor programming model as well as achieve high availability and performance, it is an essential question to decide which entities or entity sets should be modeled as actors.

To appreciate a concrete example of this challenge, consider the beef cattle system introduced in Section 2.2, where many entities perform different tasks to satisfy multiple requirements. For instance, a collar sensor that is bound to a cow continuously collects real-time geo-data for this cow and sends it to the data platform. A historical trajectory for each individual cow is thus updated based on this sensor data. Besides, additional information may be needed, such as the cow's identifier or health-related data. In this sense, we can observe both collar and cow as separate entities engaged in cooperation to provide real-time and historical geo-information to farmers and slaughterhouses. The question is whether these two entities, the collar sensor, and the cow, should be one or two independent actors.

Actors comprise a model of computation for concurrency and distribution (Agha, 1990). So not only should actors encapsulate state, as it is the case with non-actor objects, but they should also abstract concurrent tasks that need to be processed by the system. In our experience, we found it useful to answer the following questions when attempting to identify actors: (a) What services are provided by the system being modeled? (b) Who should provide these services and who are their users? (c) What is the output and input of every single task performed and can these tasks be executed concurrently?

Take our beef cattle tracking and tracing system as an example. Typically one actor is designed to carry out one specific real-world task with associated logic, such as slaughter or distribute. Different actors then capture simultaneous tasks. For instance, farmers would like to obtain information on cows and manage the herds that they own. Slaughterhouses would like to obtain information about cows that will be slaughtered and record how these cows get transformed into meat cuts. Farmers and slaughterhouses can thus be conceptualized as users of cow information services, which a cow ought to provide. Moreover, the interactions between farmers, slaughterhouses, and cows are concurrently executed by independent entities in the real world. In particular, farmers manage cows and their respective information, and slaughterhouses slaughter cows and record their transformation into meat cuts. Cows are associated with their sensor data, which are continuously updated by their

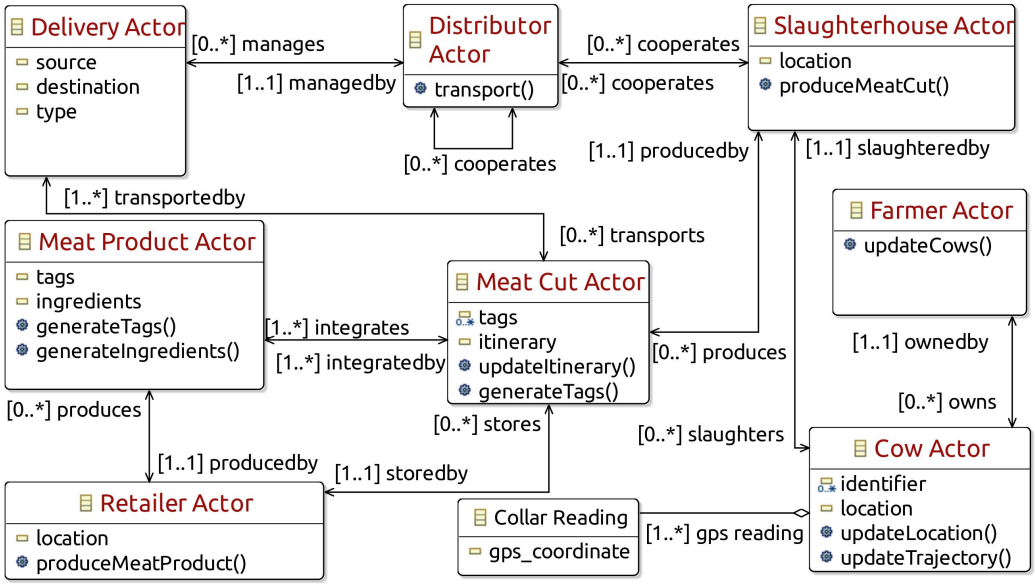


Figure 2.3: Actor Model of Beef Cattle Tracking and Tracing Data Platform.

collars. As such, we model farmers, slaughterhouses, and cows as independent actor types. Since each collar is bound to a cow, we encapsulate this sensor information inside cow actors.

Figure 2.3 shows the data platform model for the beef cattle system. Every actor encapsulates its state and communicates with other actors via asynchronous messages. Therefore, simple accesses to data in the state of an actor are rendered as asynchronous communication events across actors. As we can see from the figure, we model one cow as a *Cow* actor. A Cow actor has an aggregation relationship with many collar sensor readings indicating the GPS locations of the cow, and each such sensor reading is bound to exactly one cow. In other words, we use aggregation relationships to indicate that the objects of a non-actor class are encapsulated in the referred actors. Since cows are modeled as actors, real-time locations are reported to Cow actors, which serve this information to all interested readers along with other associated cow state data such as the cow identifier.

We model one farmer or several farmers who work together (e.g., a cooperative) as one single *Farmer* actor because the state of this farmer or these farmers is organized as a unit.[1] One cow is owned by one farm unit, but one farm unit can own many cows. The Farmer actor can read the properties of any Cow actor that is associated with it through message passing. If such

[1]Notice that this unit can be broken down into smaller units at will, depending on the focus of an application – e.g., individual farmers unite to provide beef, but a cooperative handles each farmer individually.

messages are exchanged under a security model with authentication, then we can enforce that cow information is only visible to its owner farmer tenant or properly authorized slaughterhouse.

A physical slaughterhouse is modeled as a *Slaughterhouse* actor. A cow can only be slaughtered once in exactly one slaughterhouse, but a slaughterhouse is responsible for slaughtering many cows. This constraint is reflected in the association between Cow and Slaughterhouse actors, and as above a Slaughterhouse actor can read data from any Cow actor via asynchronous messaging. The Slaughterhouse actor processes such data to derive *Meat Cut* actors, which represent units of beef to be distributed as a whole.

A Meat Cut actor processes updates to its itinerary property generated by *Delivery* actors as meat cuts are transported. In our model, a *Distributor* actor manages multiple Delivery actors, which themselves manage a transportation process with different source and destination locations. For example, a logistics company is modeled as a Distributor actor, and transportation processes in this company are modeled as various Delivery actors managed by the Distributor actor. A Delivery actor tracks a meat cut delivery from a source to a destination location using a given vehicle at a well-defined time. A meat cut can be delivered many times by one or several distributors during the whole itinerary, and a distributor is responsible for delivering many meat cuts.

We model the final destination of a meat cut to be a retailer, e.g., a supermarket chain, whose information is managed by a *Retailer* actor. Retailer actors can create *Meat Product* actors by disaggregating or combining meat cuts. Thus, Meat Product actors have a many-to-many association with Meat Cut actors.

Based on the above modeling process, we can summarize a general **principle of how to identify actors**:

> Typically, one actor is designed to carry out one specific real-world task with associated logic. Different actors capture different simultaneous tasks.

2.4.2 What should the Granularity of Actor State be?

In an AODB, we allocate different tasks into separate actors, as this organization can help increase concurrency. However, if a single actor concentrates too much state or too much of the application logic, i.e., if an actor is *coarse-grained*, then it becomes increasingly difficult to reap the benefits of concurrency stemming from the application of the actor model. At the other extreme, since actors do not share state and communicate only through asynchronous messaging, an excessively *fine-grained* actor design can introduce unnecessary overheads in state manipulation as well as increased communication overhead. Moreover, a fine-grained design may cause the actors in the system to explode in number, e.g., one actor per data item or record in the system, which can challenge efficiency in an AODB platform. So deciding the granularity of actors is an essential problem when modeling any application with actor-oriented databases.

Previously, we have formulated a principle to identify actors out of the entities in an application scenario. However, we should balance this principle against the potential effect of actor granularity on application performance. In particular, we wish to keep concurrency high, but at the same time avoid unnecessary overheads and reduce the complexity of application modeling. To balance these goals, our experience has been that it is natural to make actors more fine-grained when they represent *active entities* for which detailed tracking is required by the application.

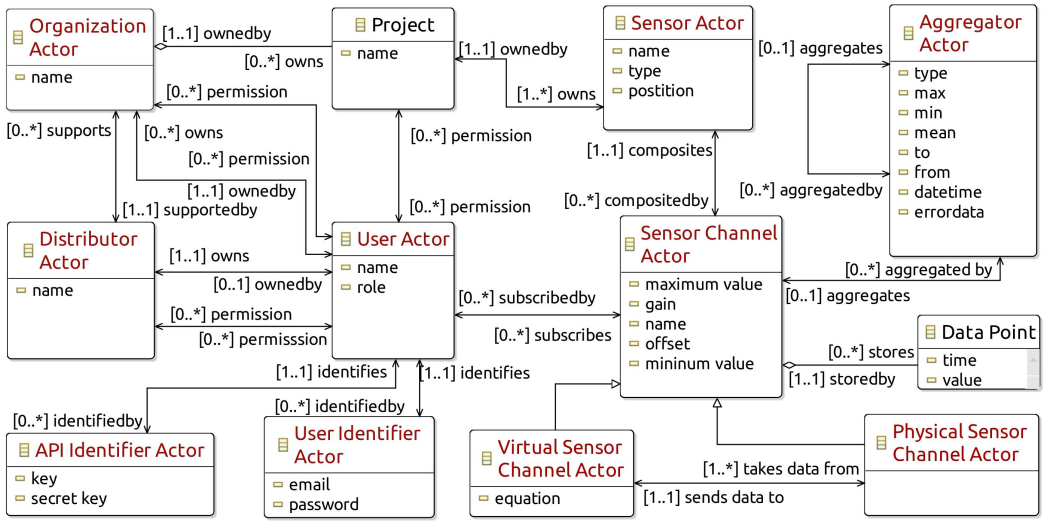


Figure 2.4: Actor Model of Structural Health Monitoring Data Platform.

Figure 2.4 presents the data platform model for the structural health monitoring system. We observed during modeling that organization entities own project entities representing different constructions and that each such construction project is associated with some installed sensors. Note that only organizations are active, as they initiate and manage construction activities, while projects are passive structural schemes used by organizations. As such, we create *Organization* actors that encapsulate project information, as displayed by an aggregation relationship with a non-actor Project class, instead of utilizing separate actors. This modeling decision minimizes message exchange when there is no clear advantage in having the two entities run concurrently.

This notwithstanding, sensors are themselves active entities in that they may be relocated, leading to change of position, and also may generate multiple data streams originating from different physical sensor channels (e.g., if we consider a regular smartphone as a sensor, then the accelerometer and microphone would be sensor channels). Moreover, messaging is minimal between sensors and sensor channels, as data streams arriving at the platform can be disaggregated by proxies directly by sensor channel instead of being relayed through sensor entities. As such, we model separate *Sensor* and *Sensor Channel* actors. Sensor Channel actors hold a window of data points originating in the respective data stream. The data points are captured as non-actor objects since these entities are not active.

To help structure information about data points, additional actors are included. First, Sensor Channel is specialized into *Physical Sensor Channel* and *Virtual Sensor Channel* actor classes. Whereas the former represents a channel in a physical sensor, the latter represents a computation over potentially multiple physical channels (e.g., in our smartphone example, an equation merging the data from accelerometer and microphone sensor channels). While a virtual sensor channel provides data at the finest level of detail, it is necessary to provide statistical aggregates for online queries posed by data analysts at various levels of detail (e.g., per hour, day, or month). Since there can be parallelism in computing these aggregations across levels of detail (e.g., hourly aggregates serving as input to daily aggregates), it is useful to conceptualize them as active entities. We thus introduce *Aggregator* actors in the model.

Based on the above modeling process in the context of our case studies, we summarize a general **principle of how to decide on actor granularity**:

An actor should represent the functionality of one active entity for which detailed tracking is required.

2.4.3 What is the Trade-Off between Employing Actors or Non-actor Objects for Frequently Accessed Entities?

We discussed the issue of actor granularity, which may result in decisions where entities from the domain are modeled as actors or alternatively as non-actor objects. The modeling principle for actor granularity calls our attention to active entities. By contrast, there are a number of entities that store data but do not proactively perform tasks. We call them *inanimate entities*, and they are exemplified in the beef cattle tracking and tracing case study by meat cuts and meat products. In Figure 2.3, we model these inanimate entities as actors. However, these actors only encapsulate state and manage corresponding queries and updates originating from active entities such as slaughterhouse, distributor or retailer, e.g., when meat cuts and products are created or transported. As such, a natural question is whether these inanimate entities could have been modeled as non-actor objects instead of actors.

For example, suppose a distributor wishes to obtain information about a meat cut that it transports. The corresponding Distributor actor would have to send a message to the respective Meat Cut actor to fetch this information. Furthermore, when a meat cut is transported, the Meat Cut actor has to communicate with a number of other source or destination actors, such as Slaughterhouse, Distributor, or Retailer actors. As such, a Meat Cut actor frequently interacts with other actors in the system. Since all information on meat cuts needs to be exchanged across actors through asynchronous messaging, casting meat cuts as actors can generate a considerable communication overhead.

To explore this question, we have created an alternative model for the beef cattle tracking and tracing case study (cf. Figure 2.5). In this alternative model, we capture inanimate, but frequently updated entities, such as meat cuts and meat products, as non-actor objects instead of actors. Actors are marked in red in Figures 2.3, 2.4 and 2.5, while the non-actor objects are marked in black. The non-actor objects represent a state and thus cannot exist in an AODB independently of some actor. To capture state mutation as meat cuts

and products move across the supply chain, we create object versions that are always associated with a responsible actor at every stage. Consider how a meat cut is transferred from a slaughterhouse to a distributor. The meat cut is the same real-world entity, but the slaughterhouse and distributor may identify the meat cut differently. Upon transfer, the object representing the meat cut will be copied from the Slaughterhouse actor to the Distributor actor, where this new object version can be updated. Since each actor keeps a separate object version of the meat cut throughout the supply chain, communication to obtain meat cut information is obviated. All the actor logic that reads this information can now access the encapsulated entities in the respective actor state. For frequently accessed entities, this reduction in communication may pay off with respect to the overhead of copying non-actor objects. Furthermore, potentially more concurrency can be exploited in reading local object versions across several actors independently. However, some degree of data redundancy may be introduced in the model.

Based on the above modeling process, we can summarize a general **principle of when to model frequently accessed entities as non-actor objects instead of actors**:

> Frequently accessed entities can be modeled as actors or non-actor objects, and the latter representation should be preferred when reductions in communication overheads and gains from concurrency offset the disadvantages of copying overhead and data redundancy.

2.4.4 How can Relationship Constraints be Enforced across Actors?

Because actors encapsulate state and only communicate through asynchronous messaging, relationships between actors are conceptually distributed. For instance, in the model of Figure 2.3, a farmer may own many cows, but a cow belongs to at most one farmer. In a typical implementation, each direction of this relationship would be represented as properties in Cow and Farmer actors. When performing updates to this relationship, we need to update both sides and make sure these two properties in different actors remain consistent. In particular, when a farmer sells a cow, the Cow actor should have its *ownership* relationship changed to the next owner, and the properties in the two affected Farmer actors ought to reflect that only one farmer retains the ownership

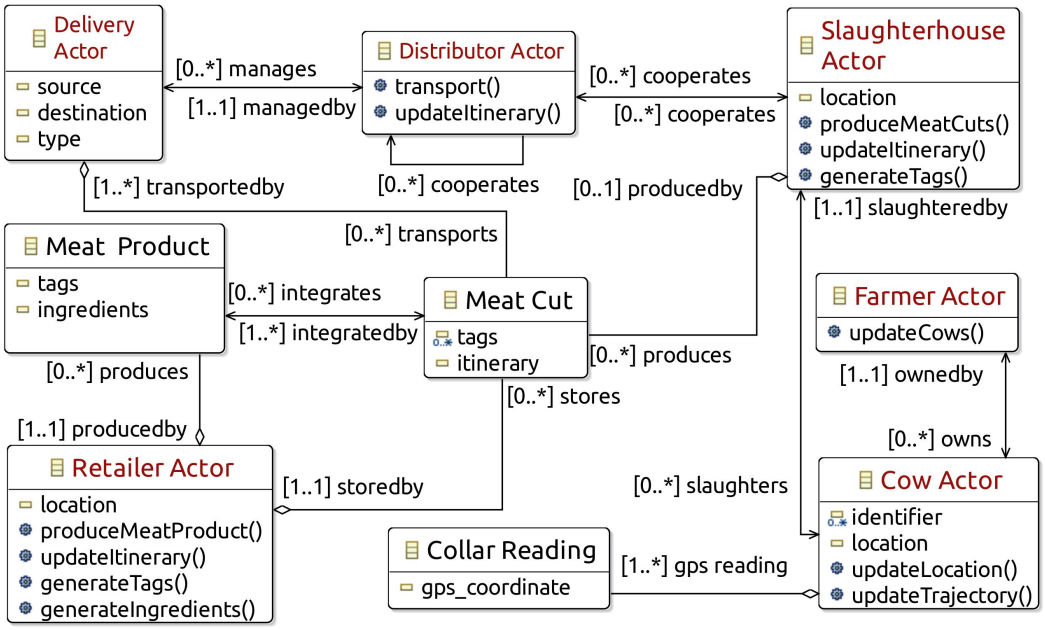


Figure 2.5: Alternative Actor Model of Beef Cattle Tracking and Tracing Data Platform.

of that cow. Since communication between actors is asynchronous, it is a challenge to keep consistency across actors in the presence of updates.

The consistency problem can be addressed by a transaction facility in the AODB, when available, or alternatively by a workflow that ensures that all actors in a relationship change are eventually updated to a consistent state. These options are similar in spirit to the proposal for indexing support in AODBs (Bernstein *et al.*, 2017b). Since some actor systems, such as Akka, no longer support transactions (*Akka Migration Guide 2.3.x to 2.4.x* 2018), and update workflows operate under relaxed consistency, a final alternative is to keep all data related to a relationship, or more generally constraint, encapsulated in a single actor. This discussion leads us to our final **principle of how to enforce constraints when using actors**:

> Employ transactions to update data across actors consistently; however, in the absence of transactions, keep data related to a constraint in a single actor or design a multi-actor workflow for updates.

2.5 Implementation

In this section, we discuss the implementation of the IoT data platform for the first case study of Section 2.2 with an AODB. We choose the Structural Health Monitoring Data Platform (SHMDP) since the resulting implementation has been transitioned to the company SenMoS. However, the lessons learned and discussion extend more broadly to the applicability of AODBs to IoT data platforms in other domains, e.g., in beef cattle supply chains, among others.

2.5.1 Choice of AODB

Our implementation of the structural health monitoring data platform was based on the model of Figure 2.4 (Borggren, 2018). The first implementation challenge to be overcome was to find an appropriate platform supporting actor and non-actor object constructs, as well as the AODB approach. The vision for AODBs (Bernstein *et al.*, 2017b) was proposed in the context of the Orleans project (Bernstein *et al.*, 2014), and we thus elect this actor runtime for the SHMDP. Orleans has also been used successfully in the context of other scalable applications (*Who Is Using Orleans?* 2018), and can thus support real-world deployments. Unlike many other actor programming languages or frameworks such as Erlang (*Erlang-Build massively scalable soft real-time systems* 2020) or Akka (*Akka - Build Powerful Reactive, Concurrent, and Distributed Applications More Easily* 2020), Orleans employs the concept of *virtual actors*, i.e., named actors that are logically in perpetual existence. The Orleans actor runtime automatically creates activations of these virtual actors for processing whenever functions are asynchronously invoked on them, and eliminates activations when there is pressure on resources. As such, virtual actors simplify actor lifecycle management for an application built on Orleans.

In addition to virtual actors, Orleans provides an explicit storage model for actor state. In particular, actors run in a stateful middle-tier that can be conceptualized as an in-memory cache of actor state enriched with application code expressed as actor functions. Whenever persistence of actor state is required, a cloud storage system is employed by Orleans. The concrete storage system is specified through annotations in actor code. To meet the vision for AODBs, additional features are currently being implemented in Orleans to close the gap between actor runtime and DBMS functionality, e.g., indexing (Bernstein *et al.*, 2017b) and ACID transactions across actors (Eldeeb and Bernstein, 2016).

2.5.2 Data Platform Architecture

A second implementation challenge was to architect an IoT data platform based on AODBs that fulfills all of the non-functional requirements of Section 2.2. Ideally, an AODB should handle online data ingestion and querying as well as analyses of historical data. However, as pointed out in Section 2.2, declarative querying functionality is still incomplete in AODBs currently (Bernstein *et al.*, 2017b). Thus, we identify three core components for the SHMDP: actor runtime, cloud storage system, and analytical database system. The actor runtime was implemented by Orleans and provides the virtual actor abstraction. It also keeps any necessary in-memory data structures for online data processing and analysis as expressed in the model of Figure 2.4. The storage system provides durability of actor state, and allows large amounts of historical data to be archived. A key-value database system with efficient data ingestion (Luo and Carey, 2018) is useful for this purpose. Finally, data recorded in the storage system can be exported into a classic star schema implemented in the analytical database (Kimball and Ross, 2013). The latter component is targeted at analytical queries over historical data, and its description is outside the scope of this work. The former two components comprised the online data ingestion, processing, and analysis functions of the SHMDP.

2.5.3 Support for Non-Functional Requirements

The AODB architecture supports the non-functional requirements listed in Section 2.2 as follows:

1. **Data ingestion from endpoints**. Data from different endpoints was managed by distinct actors in Orleans, and recorded in the cloud storage system for durability.

2. **Multi-tenancy**. Modularity, data encapsulation, and asynchronous communication were provided by virtual actors in Orleans, allowing isolation of functions and data sensitive to different users.

3. **Support for heterogeneous data**. Orleans virtual actors support a number of data types and structures, e.g., representing simple alerts or real-time derived data for virtual sensors. In addition, Orleans was used to query time ranges of raw data, and to build aggregates for low latency requests over time periods. The problem of using Orleans for these

functionalities was that declarative queries cannot access data across actors, and thus needed to be decomposed by the developer.

4. **Cloud-based deployment**. Orleans was built to scale out on servers, and extend over multiple geographical locations. It is, moreover, open-source and designed with cloud deployment as a primary target.

5. **Scalable data platform**. Modularization allows scalability in the number of actors, thus easily enabling the addition of more endpoints or users to the data platform.

6. **High efficiency**. All processing in virtual actors occurs in-memory. Orleans employs multi-core and multi-server architectures to execute application logic in different actors in parallel.

7. **Access control and data protection**. Authentication and access control were implemented at the application level by building on actor modularity features.

2.5.4 Virtual Actor Durability and Deployment

Further implementation challenges arise from ensuring that the IoT data platform can effectively ingest and process the large number of concurrent update streams originating from devices. Two issues may impact performance substantially: enforcing durability and deploying actors over multiple machines in a cloud infrastructure.

Orleans virtual actors are called grains, and managed automatically by the Orleans runtime. When a grain has work to do, the grain is activated; when the grain has been standing idle for too long, the grain's resources are reclaimed by the system, removing it from memory. To provide durability, grains in Orleans may have a state storage class. This class defines all variables the developer wishes to store persistently. The developer can force the current state to persistent storage by invoking the WriteStateAsync grain method or configure the grain class to store state persistently when Orleans deactivates a grain. Whenever the Orleans runtime re-activates a grain, the runtime retrieves by default the latest grain state from cloud storage, if available. As such, Orleans lets the developer decide when state is written to persistent state storage.

In the SHMDP, durability requirements may vary depending on the task being implemented. Certain tasks require that the state of actors be immediately

made durable, e.g., for creating structural entities such as organizations, sensors, projects, and sensor channels. Other tasks, such as gathering sensor data, can collect a window of updates before forcing them to storage. For example, in the Great Belt Bridge (*Facts and History* 2018), the structural health monitoring project consisted of more than 200 sensor channels, with a typical requirement for live data being a reporting rate of one packet of readings per sensor per second. So if we wrote state to persistent storage after each request, we would need 200 write requests every second to the cloud storage system.

Activated grains in Orleans get distributed across a set of silos, where each silo is typically deployed in a server in a cluster of machines. The distribution of grain activations to silos is by default random, which is adequate for most use cases since it will spread load. However, this actor deployment can increase the cost of communication when certain actors interact frequently. Orleans suggests using prefer-local activation in these cases. For our data platform, we have had to change the activation placement strategy away from random placement for our sensor channels and aggregators. The prefer-local placement in these instances minimizes the need to perform remote procedure calls when processing incoming requests.

2.6 Experimental Evaluation

Our goal in the experiments is to assess if the AODB-based implementation of our model from Figure 2.4 yields an IoT data platform that can scale in the number of sensors simulated and at the same time support low-latency online query functionality. In the following, we present our setup and the obtained results.

2.6.1 Setup

Benchmarking Tool

To stress-test the SHMDP, we created a command line tool in .NET that uses the Orleans framework client directly. This tool simulates data requests from sensors and users in order to generate variable load for the data platform. Sensors are simulated by tasks that each call a sensor grain and insert 10 data points. This procedure is repeated each second if all sensors have finished

their calls, so as to adhere to the behavior expected in the real scenario based on our experience.

Even though we simulate sensors for experimentation with the benchmarking tool above, we envision that ingestion of sensor data points will be based on a REST interface in a production deployment. This way, sensors can send HTTP calls to the data platform. As part of data ingestion, message queues can be employed to accommodate for bursty behavior in sensor measurements (*Azure Queue (AQ) Stream Provider* 2019). To limit the scope of our evaluation, however, we focus on stressing only the virtual actor implementation of the IoT data platform, and not other layers related to communication with sensor devices.

The benchmarking tool stores data from each request sent to the data platform in a log. Each log entry includes the latency for the request, which request was sent (data insertion, live user data, or user data request), the sampling rate, and a timestamp. With this information, we can derive detailed throughput and latency statistics for the experiments.

Summary of Software

We needed the execution of several components for the experiments. The first one was the *Orleans silo*, typically with one instance deployed per server, where virtual actors are activated and run all application logic. We also employed *Amazon RDS* (*Amazon Relational Database Service* 2018) for Orleans system storage, which keeps track of silo instances, reminders, and general system state. *Amazon DynamoDB* (*Amazon DynamoDB - Fast and flexible NoSQL database service for any scale* 2021) was used for Orleans grain state storage. Besides, the C# benchmarking tool described above is invoked to generate load to silos.

Cloud Service and Deployment

To characterize the SHMDP's data ingestion and processing capabilities, we set up our benchmark environment on Amazon AWS (*Amazon Web Service* 2020), employing the Amazon DynamoDB and RDS services (*Amazon DynamoDB - Fast and flexible NoSQL database service for any scale* 2021; *Amazon Relational Database Service* 2018) as stated above and EC2 on-demand instances (*Amazon EC2 - Secure and resizable compute capacity to support virtually any workload* 2020) for all remaining components. Given our budget, two types of instances

were employed: T2 for low cost and burst performance features as well as M5 for more stable performance. All instances were running Windows Server 2012 R2 and Orleans 1.5.0. The configuration, unless otherwise mentioned, was designed to simulate a possible future production deployment of the data platform based on our previous experience with the project for the Great Belt Bridge (*Facts and History* 2018): m5.xlarge instances were employed for the Orleans silos, RDS db.t2.small for Orleans system storage, DynamoDB with 200 writes and 200 reads per second for Orleans grain storage, and an m5.2xlarge instance for the benchmark tool.

Environment Configuration

For the experiments, we simulate sensors with two sensor channels each; every tenth sensor has a virtual sensor channel that is a summation of the two other sensor channels on the corresponding sensor. The latter choice reflects that only a subset of sensor data require additional processing to create a derived virtual sensor stream, which is close to the real life scenario from the Great Belt Bridge. We populated our actor-oriented database with synthetic data for users, organizations, projects, sensors, and sensor channels simulating a realistic scenario. For every 100 sensors, a new organization was constructed with a single user and a single project. Following the sensor configuration, these 100 sensors represent 210 sensor channels in total, out of which 200 are physical sensor channels and 10 are virtual sensor channels. This structure was used for all experiments, so that we can calculate exactly how many organizations, projects, users, and sensor channels are created given a number of sensors. Employing 100 sensors with 210 sensor channels in total is a configuration similar in size to the one in our previous experience with the Great Belt Bridge.

To achieve our experimental goal related to low latency queries, the upload of data points to the grain state storage has been configured to only happen when the Orleans silo service is shut down. This configuration ensures that we are not benchmarking DynamoDB storage, but rather the execution of in-memory actors. When using the system in production, the grains have to be configured to store data points to grain storage at an acceptable rate as explained in the considerations for durability in Section 2.5.

Load was offered to the SHMDP by sending requests with 20 data points for each sensor currently being simulated (i.e., 10 data points were generated

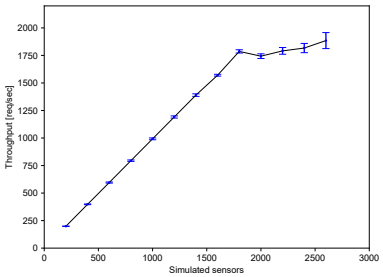


Figure 2.6: Single-server throughput experiment.

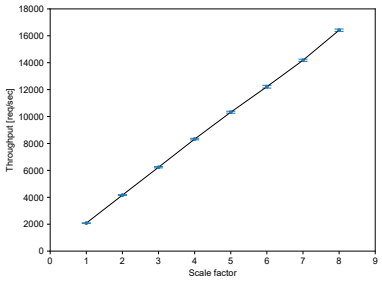


Figure 2.7: Scale out experiment over multiple servers.

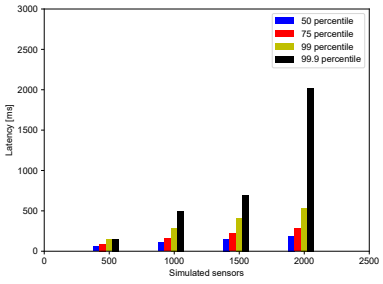


Figure 2.8: Latency percentiles for raw sensor channel data point time range requests.

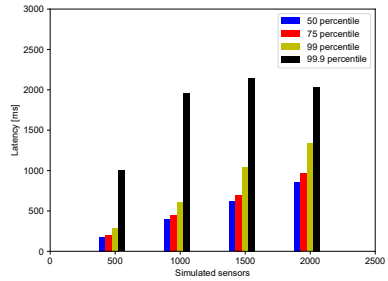


Figure 2.9: Latency percentile for organization live data requests.

for each physical sensor channel in each sensor). The requests were sent at a rate of 1 request per second. This frequency simulates sensors sampling data at 10 Hz, as specified in the Great Belt Bridge project for most sensors. As an example, consider that we wish to simulate 500 sensors: this number of sensors would correspond to 1,000 physical sensor channels and 50 virtual sensor channels. Thus, the resulting load would be of 500 requests per second being used to transmit 10,000 data points per second, and leading to the calculation of 500 virtual data points each second.

For each experiment, Figures 2.6, 2.7, 2.8 and 2.9 present the results. A single point on the figures aggregates 10 minutes of the whole service configuration running. The data was split into windows of 1 minute, and the first minute was removed to make sure the platform had started up correctly before measurement. In addition, the last minute was removed to ensure that only whole minutes were used. The average latency or throughput was then calculated as a measurement, and depicted along with standard deviation as error bars where appropriate.

2.6.2 Experimental Results

How many sensor readings can the SHMDP ingest using a single cloud server?

In our first experiment, we aimed at establishing a relationship between the number of simulated sensors and the hardware utilization at the data platform, so that we can create a baseline load for the other experiments. In particular for these measurements, we employed the smallest VM size in the M5 series, the m5.large instance, and observed when the instance cannot process any more data insertion requests. We have chosen the smallest server size so that the experiment can be used for both scale up and scale out baselines.

Figure 2.6 shows the results from this single-server throughput experiment. Because each simulated sensor in the experiment was configured to send one request every second, note that the SHMDP deployment was processing all requests as long as the throughput is equal to the amount of simulated sensors. We observe that the ratio between simulated sensors and throughput is close to one until the number of simulated sensors reaches 2,000. At that point, throughput ceases to increase even if more load is offered. By monitoring the VM instance when performing the experiment, we have remarked that CPU Usage in Windows Task Manager was at 100% when the number of simulated sensors was above 1,800.

Does the SHMDP scale simultaneously on the number of sensors and servers?

Our second aim was to verify whether the data platform can scale out in the number of data requests that it can ingest from simulated sensors by utilizing the computing power of more servers. To simulate a production environment, we employed larger m5.xlarge VMs as described in the experimental setup. From our single-server throughput experiments, we can estimate a baseline load to be offered per server. Based on this baseline, we can proportionally scale the load, number of servers, and organization structure in the experiment.

To estimate baseline load, we note that in a production environment, we wish to leave some CPU resources for user interaction. We chose to leave roughly 20% utilization for handling user online query requests and creating statistical aggregates. From the single-server throughput experiment of Figure 2.6, we

know that roughly 1,800 requests per second can be processed by a m5.large instance. By removing 20% and rounding to the nearest 100 requests per second, we obtain 1,400 requests per second. Now, we can scale that number by the difference in computing power between the m5.large and m5.xlarge instances, which is estimated by their EC2 Compute Unit (ECU) values to be of a factor 1.5x. So the baseline for a single server corresponds to the load offered by 2,100 simulated sensors. This configuration is employed for a scale factor of one. As the scale factor is increased, we proportionally increase the number of simulated sensors and the number of servers used for Orleans silos.

Figure 2.7 shows that the throughput sustained by the data platform scales close to linearly with the scale factor. To illustrate this observation, consider that at a scale factor of five, we have five server instances and 10,500 simulated sensors. We observe as expected a throughput above 10,000 requests per second. Similarly, for a scale factor of eight, we have eight server instances and 16,800 simulated sensors, and a throughput above 16,000 requests per second is observed.

The results indicate that the data platform can potentially scale out even further than the 8 servers used in this experiment, since we did not hit any bottlenecks. We expect that the behavior can be maintained as we add a larger number of servers, since there are no dependencies across organizations and there is enough processing slack left to support eventual online user queries and calculation of statistical aggregates.

Does the SHMDP deliver low latency on online query functions concurrently with data ingestion?

We have simplified the previous experiments by removing any user interactions, and made all sensors sample data at 10 Hz sending 1 request each second to the data platform. This scenario is close enough to our experience with a real deployment that we can observe how the data platform scales as we increase the number of sensor insertion requests. However, we still need to show that the 80% utilization rate chosen earlier will indeed leave enough room for the processing of online user queries. Furthermore, we aimed at better characterizing user request latencies under this target utilization level to make informed decisions when creating a production environment in the future.

To simulate user requests to the data platform, we estimated a relationship between simulated sensor requests and user interaction requests. We know from the requirements for the SHMDP that requests for live data as well as raw data kept in the sensor channel actors need to be supported. Requests for live data retrieved the most recent values from all sensor channels of a given organization, while requests for raw data retrieved the time series in a given sensor channel actor in an organization. From actual user interactions observed at the Great Belt Bridge project, we expect these online queries to be generated by at most one person looking at live data for each organization requesting data once every second, and at most one request for raw data a second for each organization. Since a deployment in that project would have around 100 sensors, we thus generate roughly 1% of the requests for live data from all sensor channels in a organization, 1% for raw data, and the remaining 98% as sensor data insertions.

Figures 2.8 and 2.9 show that the latency of online query requests increase, as expected, for higher percentiles of the latency distribution. This growth is especially pronounced for 99.9th percentile latency; however, even these extreme tail latencies can be ameliorated if utilization is reduced in the machine by offering load from less sensors. For example, for 500 simulated sensors, 99.9th percentile latency is minimal for raw data requests, and under 1 sec for live data requests. It is expected that latency of user interactions on the website be kept within a few seconds. This requirement can be fulfilled by the data platform even with the targeted 80% utilization load offered by 2,000 simulated sensors and at extreme latency percentiles. Moreover, the latency of raw data requests is often substantially below 0.5 sec, while that of live data requests is often below 1 sec at 2,000 simulated sensors.

2.7 Related Work and Discussion

This section discusses research efforts related to our work. To the best of our knowledge, the literature lacks contributions explicitly justifying why and discussing how AODBs meet the challenges of IoT data platforms. However, earlier approaches have explored how to support different aspects of IoT data management employing a variety of data-centric system abstractions.

Approaches based on data stream management systems (DSMSs), in particular, are a commonly used solution in the context of IoT systems (Shen *et al.*, 2015;

Calbimonte *et al.*, 2010; *AWS IoT Core* 2018; *Google IoT Core* 2018). DSMSs are apt at transforming multiple input streams, through a topology of data flow operators, into output streams containing, e.g., alerts and notifications for further processing. One challenge in these systems has been flexibility in responding to dynamically changing conditions as typical in IoT, e.g., through the addition or removal of input sources (Singhal *et al.*, 2018). Actor-based streaming implementations have been proposed to address these concerns (*Akka Streams version 2.5.18* 2018; *Orleans Streams - Microsoft Orleans Documentation* 2020), as adaptability is a built-in feature of the actor model (Agha, 1990). However, a problem with data streaming approaches has been to additionally provide for data storage and online queries (Chandrasekaran and Franklin, 2004; Dindar *et al.*, 2011). In the context of IoT, AllJoyn Lambda explored a lambda architecture for IoT data storage analytics. Adnan et al. combined streaming and historical data to perform predictions in IoT systems based on machine learning models (Akbar *et al.*, 2017). In contrast to AODBs, which abstract storage management with virtual actors and storage annotations, these approaches require developers to master complex APIs, often spanning across data stream and database systems. Moreover, while these systems provide for low-latency alerts, online queries are non-trivial to support efficiently. By contrast, an AODB acts as an in-memory, programmable cache where complex analyses can be executed in parallel over the encapsulated state of multiple actors employing user-defined methods.

Another class of solutions explored by previous work is that of cloud-centric actor-based IoT middleware, such as Ptolemy Accessor (*The Ptolemy Project: Accessors* 2018) and Calvin (Persson and Angelsmark, 2015). In these systems, every IoT device is modeled as an actor so that those multiple IoT components can be easily integrated into a potentially complex edge-cloud system. However, these middleware platforms lack integration with data management features that are central to an IoT data platform, such as efficient data storage with support for multi-tenancy and data protection. In addition to middleware, specific IoT applications have also been directly built over actor runtimes (*Akka Documentation, Version 2.5.17, IoT example use case* 2018; *Who Is Using Orleans?* 2018). For example, Pegasus is a cloud-based project aimed at gathering data with high-altitude balloons (Chansanchai, 2018). The system employs the Orleans actor runtime so as to simplify the development process of building a parallel, interactive and dynamic cloud service (Bernstein and Bykov, 2016). In contrast to our work, these previous implementation efforts do not provide

any insights on data modeling decisions, nor do they analyze case studies to connect requirements for IoT data platforms with the necessary support from an actor-based solution. Even though there have been explorations of how to employ actors as a modeling construct for cyber-physical systems (Derler *et al.*, 2012), none of these investigations fully satisfy our data platform requirements, namely storing, managing and processing large-scale data as well as providing for high scalability, real-time computation, data protection, and access control.

In line with the vision of Bernstein et al. (Bernstein *et al.*, 2017b), we argue that the integration of data management features into actor runtimes can help meet the increasing demand for scalable, low-latency data platforms. Recently, a relational actor programming model has been proposed for in-memory databases and realized in ReactDB (Shah and Salles, 2018a). Even though ReactDB shows that the actor model can be used to provide for low latency in databases, we did not consider it as a possible option for our data platform because it is a research prototype and currently not available for production use. Furthermore, in previous work combining actors and databases, there is no systematic review of how to model and structure IoT data platforms, nor discussion of the implementation of such IoT platforms employing an AODB approach. Our work matches the characteristics of an AODB with the requirements and challenges of IoT data platforms, showing how recent research on AODBs can be the basis for a new methodology to model and build IoT data platforms.

2.8 Conclusion

IoT systems require adequate data platforms for handling data storage, management, query and preservation. The modeling and deployment of these platforms remain an open research challenge. In this chapter, we presented a generic actor-oriented data platform modeling approach for IoT data platforms, showing how actor-oriented databases can address challenges in the management of IoT data. Our discussion of challenges and their solution was showcased via two distinct case studies, specifically systems for structural health monitoring and beef cattle tracking and tracing. Our contribution covered the detailed modeling of these two real-world case studies and presented the entities and the patterns used to represent their dynamic behavior. This was accompanied by a discussion of modeling challenges, together with our

recommendation of technologies and methodologies to address these challenges. As part of this work, we developed a prototype of a structural health monitoring system, which was transitioned to SenMoS. This prototype was validated through experiments demonstrating scalability as more simulated sensors are added as well as low latency in interactive query functions.

We believe that adopting AODBs for IoT systems can help attain the full potential of IoT by extending the reach, scalability, and maintainability of IoT data platforms. As future work, we plan to explore data integration issues in IoT data platforms modeled with the AODB approach, and devise approaches to enforce constraints in AODBs.

Implement data integration in data platform to help analysis, build and maintain an index on actors to quickly locate actors in constraints and design of transaction or workflow to keep data consistency across actors are potential paths for future work.

3

Dolphin: An Actor-Oriented Database for Reactive Moving Object Data Management

Novel reactive moving object applications require solutions to support object reactive behaviors as a way to query and update dynamic data. While moving object scenarios have long been researched in the context of spatio-temporal data management, reactive behavior is usually left to end-user implementations. However, it is not just a matter of hardwiring reactive constraints: the required solutions need to satisfy tight low-latency computation requirements and be scalable. This emerging class of applications builds on database technology, but implements substantial data management logic in the application tier. This part of dissertation explores a novel approach to enrich a distributed actor-based framework with reactive functionality and complex spatial data management along with concurrency semantics. Our goal is to better meet the needs of reactive moving object applications. Our approach relies on a proposal of the moving actor abstraction, which is a conceptual enhancement of the actor model with reactive sensing, movement, and spatial querying capabilities. This enhancement helps developers of reactive moving object applications avoid the significant burden of implementing application-level schemes to balance performance and consistency. Based on moving actors, we define a reactive moving object data management platform, named *Moving Actor-Oriented Databases (M-AODBs)*, and build *Dolphin* – an implementation of M-AODBs. Dolphin embodies a non-intrusive actor-based design layered on top of the Microsoft Orleans distributed virtual actor framework. In a set of experimental evaluations with realistic reactive moving object scenarios, Dolphin exhibits scalability on multi-machines and provides near-real-time reaction latency.

3.1 Introduction

Recent years are witnessing a growth in research in data-intensive scenarios involving mobile devices and sensors (Costa, 2018; Cao and Wachowicz, 2020). Moving objects and their trajectories have long been subject to research in spatiotemporal databases, *e.g.*, storage and indexing of trajectories, query processing or linking to semantics – such as Points of Interest. Nowadays, applications such as collaborative transportation systems (Khajenasiri *et al.*, 2017; Cui *et al.*, 2019), fleet management and traffic monitoring (Duckett *et al.*, 2018; Albani *et al.*, 2017; McLellan, 2020; Shorinwa *et al.*, 2020; Honkote *et al.*, 2020; Atten *et al.*, 2016), tracking of mobile objects and crowd sensing platforms (Capponi *et al.*, 2019), *etc.*, concern objects that continuously move in space and react to their surroundings. These highly relevant and emerging mobile Internet-of-Things scenarios require data management frameworks to support reactive behavior and to query and update large amounts of dynamic spatial data.

Consequently, new requirements regarding reactive features for those objects are emerging. For example, such objects are sensitive to their surroundings – in particular, to the objects moving around them, – *e.g.* changing their states and sharing information to affect other "surrounding" moving objects. We name such objects *reactive moving objects* and these applications *reactive moving object applications* – to characterize the emphasis on surroundings-sensitive object behaviour, and the collective participation of such objects in a spatially-sensitive decision-making process.

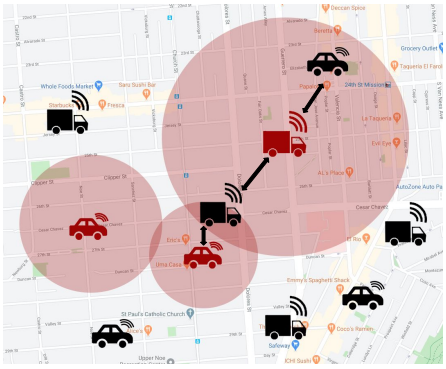


Figure 3.1: C-ITS Example

Running Example. In Cooperative Intelligent Transportation Systems (C-ITS) (Mitsakis *et al.*, 2020; Bussche, 2020; Nguyen *et al.*, 2020), vehicles communicate and cooperate towards improving overall transportation effectiveness, driving behaviours, security and safety (Ni, 2016). While regular ITS (Li and Nashashibi, 2013; Hu *et al.*, 2020; Tak *et al.*, 2020) concentrate on local information sharing, optimization and coordination (*e.g.*, involving drivers, traffic controllers, or transportation system operators) to make better decisions, in C-ITS, global infor-

mation ingestion is required and vehicles themselves are active participants in transportation actions and decisions. Figure 3.1 presents an example of C-ITS. Reactive vehicles (shown as red vehicle icons) start reactive sensing within a specific spatial region (shown as a semi-transparent red range). A reactive vehicle is aware of other vehicles if those vehicles' movements satisfy its predefined spatial predicates (*e.g.*, cross, cover, or overlap) against its spatial sensing area. Sensing areas and predicates may vary from vehicle to vehicle. In a simplified way, vehicle behavior in standard ITS is managed by the application, whereas vehicles in C-ITS are active participants in deciding their movements. Thus, a reactive vehicle can take reactive actions, such as communication with approaching vehicles to send warnings to help them avoid hazards (Uhlemann, 2018). We envision cooperation will be a critical ingredient for self-driving or assisted-driving.

Three-tier architecture, which decomposes user interface, application logic and data management into three separate tiers, is the predominant architecture for client-server systems (Kambalyal, 2010). However, as pointed out by Salvaneschi and Mezini (2013) and Bonér *et al.* (2014), a reactive system needs to update and react to the changes of inputs correspondingly in tight real-time computation constraints. This is challenging to meet in data intensive settings. In conventional multi-tier distributed data architectures, frequent data shipment is required between the stateless application tier running the application codes and the database tier storing the application state. In addition, system performance would be constrained by the network bandwidth and latency.

One might suggest to push the whole logic of a reactive moving object application down into a spatial DBMS. However, reactive features for moving objects are lacking in these systems. Trigger systems are the primary reactive abstraction available, but they are riddled with complex semantic issues for developers (Widom and Ceri, 1996). Furthermore, there are concerns regarding trigger scalability (Hanson *et al.*, 1999), which are only made worse in highly dynamic and distributed environments.

Distributed in-memory application architectures have proved to be a viable solution to address the aforementioned performance challenges (Agha, 1990; Bernstein and Bykov, 2016). These architectures often leverage the actor model as a programming abstraction for middle tier implementation (Hewitt, 2010; Karmani and Agha, 2011), while employing distributed DBMS for selective persistence of actor state (Bernstein *et al.*, 2014). Application

logic and data are encapsulated into actors, each of which naturally represents the digital twin of a real-world object (Wang *et al.*, 2019b; Bernstein, 2018). The decomposition of application logic and data into actors, as well as the distributed and concurrent actor model execution enable efficient use of computational resources to boost scalability.

In light of the success of the actor model in building high-throughput and low-latency applications, we envision it as a compelling implementation abstraction for the application tier in reactive moving object applications – each reactive object, as the digital twin of a real-world physical reactive moving entity, can be modeled as an actor. However, present actor runtimes do not have any built-in support for representing movement of objects in space, for querying their locations, or, most importantly, for specifying reactive behaviors to be triggered in response to the movement of other objects. Developers must face a choice between either: (a) scalably implementing these challenging reactive and spatial functionalities in the application tier; or (b) weaving together a complex web of systems, *e.g.*, spatial streaming platforms, spatial DBMS, and distributed mid-tier frameworks, with little support to navigate the trade-offs in cross-system data consistency semantics.

In this work, we investigate how to design and build an actor-based data platform with high-throughput, low-latency, and reactive spatial data management functionalities. Our solution supports the development of application tiers for large-scale reactive moving object data intensive applications. Our platform alleviates the burden on application developers by allowing them to concentrate on their application logic instead of being distracted by complex spatial data management tasks. In line with the terminology in (Bernstein, 2018), we call such a platform a Moving Actor-Oriented Database (M-AODB).

There are several major challenges to achieve this goal. First, it is necessary to provide a proper abstraction for representing reactive moving objects. In this sense, the classic actor model needs to be enriched with features required by reactive moving object applications, including geo-referenced attributes, spatial indexing and querying, spatial-driven reactive behaviors, etc. Second, in the actor model, actors update and query data in a concurrent and asynchronous manner. In our context, concurrency semantics have to be defined, not only to provide proper data consistency guarantees to developers, but also to achieve trade-offs between data latency and consistency. Third, these additional features need to be offered in a manner that preserves the benefits

of scalability and low latency that actor runtimes exhibit for conventional, non-spatial applications. A fourth major challenge is the coherent design and implementation of a M-AODB that facilitates distributed deployment of reactive moving object applications in the cloud.

Our research addresses these challenges through Dolphin, a concrete implementation of an M-AODB. To the best of our knowledge, Dolphin is the first in-memory spatial data platform that can be deployed on the cloud in a distributed fashion *and* supports scalable and reactive spatial data management for reactive moving object applications.

The major contributions of this part of this chapter thus include the following:

- In Section 3.2, we propose a *Moving Actor* abstraction, which brings together the actor model with reactive spatial data management. Moving Actors provide a natural and easy-to-understand approach for developers to construct scalable applications and manage reactive spatial data at the application tier. Moreover, we propose a novel architecture named *Moving Actor-Oriented Databases (M-AODBs)* that supports the proposed moving actor model. In this context, a M-AODB is a reactive, stateful, and scalable distributed actor-oriented data platform, which supports managing spatial data updates, conducting spatial queries, and processing reactive actions of moving actors.

- In Section 3.3, based on the analysis of reactive moving object application use scenarios, we define and implement two concurrency semantics for the spatial data of reactive moving actors, namely *Actor-Based Freshness semantics* and *Actor-Based Snapshot semantics*. They are essential to provide concurrency semantics for developers to reason about correctness and performance, as well as to achieve trade-offs between data latency and consistency.

- In Section 3.4, we present *Dolphin* – our implementation of the conceptual proposal of M-AODBs. We adopt a non-intrusive approach and implement Dolphin as a software library on top of the virtual actor framework Microsoft Orleans (Bernstein *et al.*, 2014). The latter allows Dolphin to leverage advantages of Orleans as a substrate – such as being cloud-ready, based on a popular object-oriented programming language (C#), and battle-tested on production services – as well as to be compatible with other projects in

the Orleans ecosystem. In addition to Dolphin be scaled over multiple cores in a single machine, it runs over multiple machines in the cloud. In Dolphin, the functionalities of reactive spatial data management were implemented using actors over space partitions to achieve scalability. We present how these actors interact asynchronously to perform data management under Freshness and Snapshot concurrency semantics.

- In Section 3.5, we conduct experimental evaluations on synthetic and realistic datasets to showcase the capabilities of our design and implementation of M-AODBs. Our results show that Dolphin supports low-latency reactions, frequent data updates and queries, and scales over multiple machines in a distributed setting.

Related work is discussed in Section 3.6, and we conclude this part of dissertation in Section 3.7.

3.2 Towards Moving Actor-Oriented Databases

3.2.1 Design Objectives

Our goal is to develop a data platform for application tier development for the aforementioned novel reactive moving object applications. We summarize the high-level design objectives as follows:

O1. Support Non-reactive Spatial Data Management. Conventional spatial data management functionalities such as data updates, spatial indices and queries, and maintenance of spatial static integrity constraints are necessary to build these applications.

O2. Support Spatial Reactive Behavior. Reactive objects must be able to sense their surroundings or other areas of space to perform reactive actions. This requires efficient processing of reactive actions of every object triggered by its surrounding peer objects in the environment.

O3. Support Concurrent Heterogeneous Moving Objects. To achieve low latency, the updates and reactive operations of moving objects should be executed concurrently. The data platform should provide a concurrent programming abstraction. Moreover, different moving objects could have customized

and heterogeneous behaviors. So the abstraction should allow modeling objects by their types and behaviors.

O4. Provide Scalability and Elasticity. The platform should scale to arbitrary numbers of objects while supporting the processing of complex space-dependent application logic. Additionally, a reactive moving object is a digital twin of a real-world entity. Its lifecycle should be the same as that of the corresponding real world entity as it enters and/or leaves the system. Since load may vary dynamically depending on the relative positions of moving objects, the system should have the possibility to manage underlying resources elastically.

3.2.2 The Moving Actor Abstraction

To achieve the above objectives, we start by looking at the programming abstraction of our data platform. Among the programming abstractions of reactive applications, the actor model represents a distinctive choice (Hewitt, 2010; Karmani and Agha, 2011; Gupta, 2012). In particular, the *virtual actor* abstraction introduced by Bykov *et al.* (2011), considers actors as modular and stateful virtual entities in perpetual existence, which facilitates a one-to-one mapping to moving objects in our context. Since virtual actors always exist, they do not require management of actor lifecycle and failures, which enhances developer productivity. Virtual actors are a natural fit to meet the scalability, availability, and elasticity challenges of emerging reactive moving object workloads, since they can be automatically and dynamically activated or deactivated and replicated to handle demand while balancing load across servers (Bernstein, 2018; Bernstein *et al.*, 2014). Moreover, the concerns of fault tolerance and elasticity are managed by the runtime, which makes virtual actors a very intuitive and developer friendly programming abstraction. The widespread deployment of the virtual actor runtime *Microsoft Orleans* in a variety of highly available and scalable low-latency production cloud services highlights the popularity of the programming model as well as the performance and maturity of the system (Bernstein *et al.*, 2014).

Therefore, we consider virtual actors to be a promising abstraction for managing reactive moving object data, as demonstrated by its successful and wide deployment in reactive applications. However, the actor model does not provide features that we require: (a) support for geo-referenced attributes and defining spatial and reactive functionalities in addition to user-defined

methods; and (b) reacting to actions from other objects in space or information from environment, *e.g.*, updating its state or building connections with other moving objects. To fill this gap, we propose the novel abstraction of *moving actors*, which integrates the actor programming model with reactive moving object features.

Moving Actor Formalization

An *actor* is a computational entity that keeps its private state and can only modify other actors' states by communicating via immutable asynchronous messages (Agha, 1990). To extend the actor model with the two required features above, we formalize a moving actor as follows:

Definition 1. A **Moving Actor** $a\ (id, P, M)$ is an actor comprising of the following characteristics:

1. A unique identity id to identify the moving actor a;
2. Properties P of a containing spatial information, namely: (a) the current known location l of this moving actor, where $l = (x, y) \in \mathbb{R}^2$,[1] and (b) a fence f around location l representing a polygon to define its spatial sensing boundaries.
3. Methods M including:

- `Move(`l_d`)`: Updates the location property of a from its current location l to a next location l_d. The movement of a generates a corresponding new fence f that moves along with it. Besides, a's movement may trigger reactive functions in other moving actors upon satisfaction of the spatial predicates associated to their fences (see below).
- `FindActors(`q`)`: Given a spatial query q, returns the actors that satisfy q. As a proof of concept, we focus on spatial range query. Here, q contains a spatial range r, where r is a regular quadrilateral, s.t. $r = \{\ (x_{min}, y_{min}), (x_{max}, y_{max})\ \} \in \mathbb{R}^2$. This method can be extended with other types of spatial queries, such as KNN.
- `StartReactiveSensing(`p`,` m`)`: Enables the sensing behavior of the moving actor. a calls $StartReactiveSensing(p, m)$ to start sensing the environment using a spatial predicate p, e.g., cross, cover, overlap (Clementini and Di

[1] $\mathbb{R}^2$ refers here to a two-dimensional real-valued space under a projected geospatial coordinate system, while x, y are respectively latitude and longitude.

Felice, 1995), which tests whether any moving actor's itinerary crosses, is covered by, overlaps the f of a respectively. If the movement of another moving actor satisfies p with respect to a's fence f, an application-defined reactive method m in a is invoked; m encodes the reactive behaviors of a.

- `EndReactiveSensing()`: Disables the environment sensing behavior of the moving actor.

Moving Actors in the C-ITS Example

In C-ITS, each vehicle i can be modeled as a moving actor a_i. Accordingly, properties $a_i.P$ contain the current known location of the vehicle and its fence to enable it to sense and react to the movement of other nearby vehicles. $a_i.P$ can be further augmented with application-specific properties, such as the vehicle's weight or class.

As vehicle i moves in the physical world, a_i.*Move*(l_d) is called to update its location and fence. This movement may trigger reactive functions of other moving actors. Moreover, while i moves, it may have a series of complex behaviors, depending on the spatial events on its path. Consider the scenario where the vehicle finds a hazard (e.g., aquaplaning) as it goes along a road. Not only does it want to warn approaching vehicles, but also those that it will encounter later along its trajectory. This will require not only reactive behavior in the present moment (when it finds the hazard), but also in the future over a period of time. This can be achieved by taking advantage of the reactive sensing feature with reactive methods encoding the desired reactive behavior.

If the reactive behavior of i is to warn vehicles in the hazard's proximity, a_i can invoke *FindActors*(r) to find out which other vehicles are already in a spatial range r around it and directly send a message to the corresponding moving actors about the hazard – a single reaction to its present spatial context. a_i.*StartReactiveSensing*(p,m), on the other hand, can be used for subsequent future (and continuous) behavior. Here, p is a cross spatial predicate to identify other vehicles cross i's path as it continues moving forward. Whenever this happens, m will notify the corresponding digital twin about the hazard. Finally, when i is too far from the hazard for the warning to be effective, a_i.*EndReactiveSensing()* can be invoked.

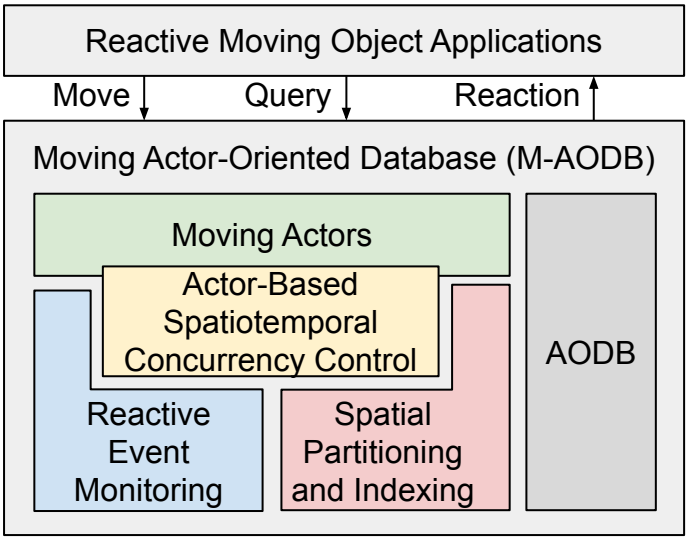


Figure 3.2: High-level Architecture of an M-AODB

Moving Actors vs. Moving Objects

It is noteworthy to point out the differences between the concepts of moving actor and moving objects as in spatiotemporal databases. Moving objects can be seen as data (Brinkhoff, 2002), over which operations are conceptually updates or queries from front-ends to back-ends on database representations of the moving objects. By contrast, moving actors compose data, operations, and reactive behaviors. They are not only used for moving object data registration and retrieval, but also to encode independently running digital twins of moving objects. As a key aspect, reactive moving objects modeled as moving actors interact with their surrounding objects and execute reactive functions themselves. For instance, in C-ITS, every individual vehicle can be modeled as a moving actor, which maintains its current location and can send queries to get information on other moving actors. Also, as exemplified in Section 3.2.2, moving actors can take reactive actions based on other moving actors' behaviors.

3.2.3 Architecture of an M-AODB

To achieve our design objectives, we build upon the concept of Actor-Oriented Databases (AODBs) (Bernstein, 2018), and propose *Moving Actor-Oriented Database (M-AODB)*, a data platform for stateful, scalable, elastic, and distributed reactive moving object data management. Figure 3.2 depicts the conceptual architecture of an M-AODB, highlighting its key components, which we describe below.

AODB

To leverage all the advantages of an AODB, an M-AODB utilizes an AODB as a building block, and enriches the virtual actor abstraction to realize moving actors. We present one possible concrete realization of this integration in Section 3.4.2, wherein the M-AODB's functionality is built into the virtual actors of AODBs.

Moving Actors

Reactive moving object applications interact with M-AODBs by leveraging the abstraction of moving actors (Definition 1, Section 3.2.2). Application-defined actors extend moving actors in our abstraction, enabling them to process moves, spatial range queries, and reactive function invocations, termed henceforth *reactions*. For brevity, the data transfer stage from physical entities to their application-defined digital twins is not included in our architecture and further discussion.

Actor-Based Spatiotemporal Concurrency Control

Moves, spatial range queries, and reactions in M-AODBs are invoked concurrently by distinct distributed moving actors. Since the logic in each actor is conceptually isolated, actor-based spatiotemporal concurrency control semantics is needed to clearly specify the visibility of movement of other moving actors within each moving actor operation. Building on the moving objects literature, we develop two such actor-based semantics, described in Section 3.3.

Reactive Event Monitoring

A move operation by a moving actor may dynamically generate reactions on other moving actors that are currently observing the spatial region where the movement took place. In M-AODBs, this is achieved by reactive event generation and matching. Section 3.4.3 gives an overview of the mechanism used in our implementation.

Spatial Partitioning and Indexing

The spatial data systems literature has shown repeatedly that spatial awareness is a pre-requisite for achieving scalability behavior over spatial workloads (Eldawy and Mokbel, 2015). In M-AODBs, spatial partitioning can be leveraged

to drive inter- and intra-machine parallelism in move and reaction processing as well as – combined with spatial indexing – to support spatial range queries (as we verify for our implementation in Section 3.5).

Using the components described above, M-AODBs inherit the benefits of AODBs and extend it with support for spatial data management and reactivity.

3.3 Actor-Based Spatiotemporal Concurrency Semantics of M-AODBs

In an M-AODB, moves, queries, and reactions take place concurrently in distinct distributed moving actors. Even though moving actors can be seen as isolated processes, their operations affect each other due to data dependencies. Therefore, a semantics needs to be defined to guarantee the correct result of concurrent operations on moving actors. M-AODBs need to decide at which identified points of time to make actor updates visible. Based on the analysis of reactive moving object application use scenarios, we specify and implement two concurrency semantics, *Actor-Based Freshness semantics* and *Actor-Based Snapshot semantics*, which are offered by M-AODBs and recommended on application level.

3.3.1 Actor-Based Freshness Semantics

The Freshness semantics, originally proposed by Šidlauskas *et al.* (2012), always provides fresh results of moving objects. Due to the asynchronous messaging in M-AODBs, we adapt it to provide “fresh" (i.e., with a recency guarantee) results of moving actors. More precisely, we define the Actor-Based Freshness semantics as:

Definition 2 (Adapted from (Šidlauskas *et al.*, 2012))**.** For a range query $FindActors(r)$ concurrently executed with movement $Move(l_d)$, assume the processing of the query lasts from t_s to t_e, the movement update completion time of a moving actor a in an M-AODB is t_u, and a moves from l_s to l_d. Assume further for any a that it can only be updated at most once during $[t_s, t_e]$.[2] The

[2]This assumption is made in (Šidlauskas *et al.*, 2012) to avoid keeping a history of move locations similarly to MVCC.

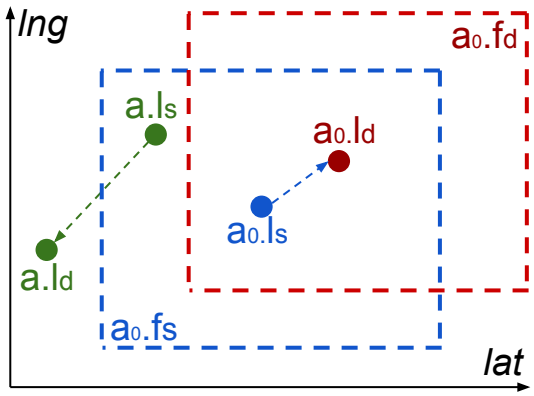


Figure 3.3: Actor-Based Freshness semantics for StartReactiveSensing

result A of $FindActors(r)$ satisfies the Actor-Based Freshness semantics if, for any moving actor a, the following holds:

- if $a.t_u < t_s$ then $a \in A$ if and only if $a.l_d \in r$.
- if $t_s < a.t_u < t_e$ then:
  - if $a.l_s \in r$ and $a.l_d \in r$, then $a \in A$.
  - if $a.l_s \notin r$ and $a.l_d \notin r$, then a $\notin A$.
  - if $a.l_s \in r$ and $a.l_d \notin r$, then a may or may not belong to A.
  - if $a.l_s \notin r$ and $a.l_d \in r$, then a may or may not belong to A.

Query results under Actor-Based Freshness semantics are dependent on the query processing time. That is because when queries are being processed, updates are also carried out concurrently, so the query results might or might not contain such concurrent updates depending on when the data are accessed by the query executor. Assume the processing of a query starts at t_s and ends at t_e. With the Actor-Based Freshness semantics, query results would reflect all updates of moving actors that have been completed before t_s, and some of the fresher updates of moving actors that are completed between t_s and t_e.

While there is no prior semantics defined for reactive actions in reactive object data management systems, we extended Actor-Based Freshness semantics to reactive behavior in M-AODBs. Figure 3.3 presents the situation that moving actors a_0 and a are in a two-dimensional space $\mathbb{R}^2$, where a_0 is defined with a continuously updated fence f under spatial predicate p and a reactive method m. After a_0 issues $StartReactiveSensing(p, m)$, if any other moving actors satisfy spatial predicate p against its fence, method m of a_0 will be triggered.

For example, p can be set to detect if any moving actor a's movement $cross$ a_0's fence. If true, method m is triggered to build a connection with a and to share the information a_0 has obtained. We provide a formal definition for $StartReactiveSensing(p, m)$ under Actor-Based Freshness semantics as follows.

Definition 3. Given $a_0.StartReactiveSensing(p, m)$ is invoked, assume the latest move of a_0 is at $a_0.t_u$ from $a_0.l_s$ to $a_0.l_d$. The sensing fences of a_0 before and after this move are $a_0.f_s$ and $a_0.f_d$, respectively. Assume further that for any moving actor a, a move is at $a.t_u$ and the itinerary of this move is $a.iti$={$a.l_s$, $a.l_d$}. Analogously to Definition 2, we assume for any such a_0, it can only be updated at most once during the time needed to process a move of a to eliminate checking predicate over multiple fences.[3] A reactive action satisfies Actor-Based Freshness semantics if the following holds:

- When $a_0.t_u <= a.t_u$, the reactive method m in a_0 is triggered if $a.iti$ satisfies spatial predicate p against $a_0.f_d$,
- When $a_0.t_u > a.t_u$,
  - the reactive method m in a_0 is triggered if $a.iti$ satisfies spatial predicate p against $a_0.f_s$ and $a_0.f_d$.
  - the reactive method m in a_0 will not be triggered if $a.iti$ does not satisfy spatial predicate p against neither $a_0.f_s$ nor $a_0.f_d$.
  - the reactive method m in a_0 may or may not be triggered if $a.iti$ satisfies spatial predicate p against only $a_0.f_s$ or $a_0.f_d$.

3.3.2 Actor-Based Snapshot Semantics

As mentioned earlier, an M-AODB is an asynchronous distributed system. In order to satisfy applications with operations on a global state of an asynchronous distributed system, we define a Actor-Based Snapshot semantics in M-AODBs. Under this semantics, a global snapshot of the system at an approximate point in time is given. In contrast to the Actor-Based Freshness semantics, where the

[3]The given assumption is not guaranteed in realistic reactive moving object applications setting, but it is true for most real cases. In the C-ITS example, given a fast speed of moving object is 80 km/h and a required high accuracy of 5 meters, move time is around 225 milliseconds, which is longer than the processing time of move in an in-memory setting. Relaxing this assumption is an extension that we leave to future work.

length of a query affects the time difference between locations returned, the results of operations depend only on the snapshot, thus making the processing time of operations irrelevant. The snapshot of the distributed application is updated periodically.

We utilize loosely synchronized clocks to facilitate checkpoint coordination in M-AODBs. In single data center cloud distributed nodes, loosely synchronized clocks differ at most by an acceptable skew (Adya *et al.*, 1995), which can be enforced by the Network Time Protocol (Mills, 1992). We argue that this is acceptable for reactive moving object applications, since minor differences in real time cannot be effectively observed in physical reality.

In M-AODBs, moving actors store update information locally. The loosely synchronized clock in each moving actor triggers its local action of exposing local state at approximately the same time to build a global snapshot in the distributed system (Adya *et al.*, 1995; Du *et al.*, 2013). Actor-Based Snapshot semantics, outlined in Figure 3.4, is defined as follows:

Definition 4. Assume the time of starting and completing the construction of a snapshot S_n is denoted by $S_n.t_i$ and $S_n.t_j$, respectively. S_n should start being constructed after the construction of the last snapshot S_{n-1} is finished, where $S_{n-1}.t_j \leq S_n.t_i$. S_n completely replaces S_{n-1} at the time $S_n.t_j$. For any moving actor a, assume a moves to l_d at time t_u. The snapshot S_n satisfies the following:

- $a.l_d \notin S_n$ if $a.t_u \geq S_n.t_i$.
- $a.l_d \in S_n$ if $a.t_u < S_n.t_i$.

Given a range query $FindActors(r)$ starts at time t_s, the result of $FindActors(r)$ satisfies the Actor-Based Snapshot semantics if the following holds:

- if $t_s \leq S_n.t_i$, $FindActors(r)$ is executed over S_{n-1}.
- if $S_n.t_i < t_s \leq S_n.t_j$, $FindActors(r)$ is executed over either S_{n-1} or S_n.
- if $S_n.t_j < t_s$, $FindActors(r)$ is executed over S_n.

In addition to formalizing the Actor-Based Snapshot semantics as above, we also extend it to reactive actions in M-AODBs. Figure 3.5 illustrates two moving actors a_0 and a in a two-dimensional space $\mathbb{R}^2$, where a_0 is doing reactive sensing with fence f, spatial predicate p and reactive function m. An M-AODB

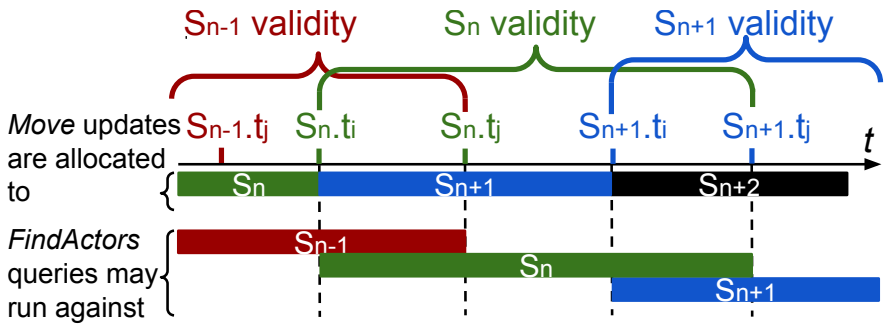


Figure 3.4: Actor-Based Snapshot semantics: Queries read from active snapshot as of their start time. Updates are guaranteed to be reflected if they precede the start of the construction of the next snapshot.

periodically updates snapshots and a snapshot S_n starts being generated at $S_n.t_i$. The accumulated spatial fence of a_0 is a minimum convex polygon covering all the fences of all the locations of a_0 between $S_{n-1}.t_i$ and $S_n.t_i$. In other words, it is the convex hull of the union of all the fences associated with the locations in the itinerary of a_0, denoted as $a_0.AccumulatedFence^{S_n}$. A moving actor a's accumulated itinerary since the last snapshot start is $a.iti^{S_n} = \{a.l_1^{S_n}, ..., a.l_{d-1}^{S_n}, a.l_d^{S_n}\}$. The location $a.l_d^{S_n}$ is the latest update location before snapshot S_n updates begin; it is also the first location for next snapshot S_{n+1}, shown as $a.l_1^{S_{n+1}}$. The spatial predicate p of a_0 is satisfied when $a.iti^{S_n}$ meets p against $a_0.AccumulatedFence^{S_n}$. That is, we check in this example whether $a.iti^{S_n}$ crosses the $a_0.AccumulatedFence^{S_n}$. In summary, the result of $StartReactiveSensing(p, m)$ satisfies Actor-Based Snapshot semantics if the following holds:

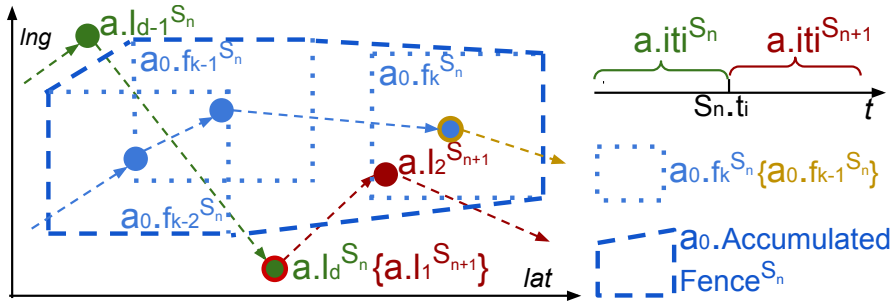


Figure 3.5: Actor-Based Snapshot semantics for StartReactiveSensing

Definition 5. The reactive method m in a_0 is triggered once at snapshot S_n if and only if $a.itin^{S_n}$ satisfies spatial predicate p against $a_0.AccumulatedFence^{S_n}$.

Actor-Based Freshness semantics vs. Actor-Based Snapshot semantics

Actor-Based Freshness semantics provides the most recent updates in M-AODBs. However, under Actor-Based Freshness semantics, query processing time affects

the amount of uncertainty in terms of location update time in the results. In general, the longer the query processing time, the more uncertain it becomes when the location updates in the results are carried out wrt. each other. Actor-Based Snapshot semantics is needed when sufficiently close to point-in-time results are needed. Actor-Based Snapshot semantics provides an image of the moving objects' locations where different locations can be contrasted with each other as they are from roughly the same time. However, these self-consistent results are achieved by staling updates. The results from Actor-Based Snapshot semantics could be stale and the construction of the snapshot is expensive, but the generated results are not affected by the query processing time. Therefore, for complex queries that take a long time to process, or queries that can tolerate data staleness but not data inconsistency, snapshot semantics is recommended. For instance, the calculation of object density would likely prefer stale results over a point-in-time snapshot, rather than fresh results over inconsistent location data. Actor-Based Snapshot semantics is also adequate for inter-object distance queries asking for the distance between pairs of objects at a point in time, because returning results with object locations at different time points would result in incorrect distances. In short, the type of semantics should be chosen based on the specific application scenarios, depending on whether the relaxed Actor-Based Freshness semantics is acceptable for the queries and reactions in the application.

3.4 Dolphin: Design and Implementation

Dolphin is our prototype implementation of an M-AODB that supports the moving actor abstraction. We design and implement Dolphin as a library to extend *Microsoft Orleans* (*Orleans Microsoft - Microsoft Orleans Documentation* 2021), which is a virtual actor programming framework for building robust, distributed systems in the cloud. Orleans provides the high-level programming abstraction of virtual actors while handling the complex responsibilities of actor lifecycle and cluster management. The design of Dolphin maintains all the benefits of Orleans while enabling the programming abstraction of moving actors. Even though we built Dolphin using Orleans, the programming abstraction of moving actors is not tightly coupled to Orleans and can be implemented using any other system infrastructure. We chose Orleans because of the maturity of the system and the convenience of the high-level programming model, which makes it a natural starting point for the design of an M-AODB.

3.4.1 Moving Actor API in Dolphin

The actor-oriented database vision (Bernstein, 2018) advocates pluggability of various database features, e.g., transactions, indexing, or streaming, to allow the application to choose and use the required features. Since C# does not support multiple inheritance using classes, we support pluggability of the moving actor abstraction by exposing it conceptually as a `mixin` using C# `interface`. An application can define a moving actor by instantiating the moving actor mixin as a property in the virtual actor thus allowing it to freely compose or inherit any class that it needs to.

IMovingActorMixin (cf. Listing 3.1a) declares the core moving actor operations described in Section 3.2.2. MovingActorMixin (cf. Listing 3.1b) is a class that implements IMovingActorMixin and provides default implementations for all operations in the IMovingActorMixin interface.[4] These default implementations cover the design and workflows described in Sections 3.4.2 and 3.4.3.

```
1 public interface IMovingActorMixin {
2     Task Move(Point l_d);
3     Task <List<ActorInfo>> FindActors(Envelope q);
4     Task StartReactiveSensing(Predicates p, Func<ReactionInfo,
      Task> foo);
5     Task StopReactiveSensing();
6 }
```

```
1 public class MovingActorMixin : IMovingActorMixin {
2     async Task IMovingActorMixin.Move(Point l_d){...}
3     async Task<List<ActorInfo>> IMovingActorMixin.FindActors(
      Envelope q){...}
4     async Task IMovingActorMixin.StartReactiveSensing(
      Predicates p, Func<ReactionInfo,Task> foo){...}
5     async Task IMovingActorMixin.StopReactiveSensing(){...}
6 }
```

Listing 3.1: (a) top: moving actor interface in Dolphin. (b) bottom: default implementations of interface.

Listing 3.2 shows in a simplified manner how a C-ITS example is implemented in Dolphin. An cooperative intelligent vehicle is a moving actor, but also includes additional capabilities, such as sending warnings to other vehicles

[4] We could alternatively employ default interface methods in C#, albeit at the cost of forcing users of Dolphin to commit to C# 8.0 or later.

who are approaching it when a hazard happens. IVehicle (cf. Listing 3.2a) is an application-defined interface for such cooperative intelligent vehicles. IVehicle extends IMovingActorMixin and further defines native functionality of vehicle moving actors. Vehicle (cf. Listing 3.2b) derives from the framework class Grain and implements IVehicle. Note that it also reuses the default implementations provided by MovingActorMixin. The methods `ReactToHazard` or `ReactToFireTruck`, for example, can be supplied as the application-defined reactive method m in a call to `StartReactiveSensing` as per Definition 1. The attribute [SpatialPreferPlacementStrategy] is our spatial-preference grain placement strategy, which enables spatial partitioning across Orleans silos (cf. Subsection 3.5.2).

```
public interface IVehicle : IGrainWithGuidKey,
    IMovingActorMixin {
     //application-defined functionality
     Task UpdateHazardInfo(HazardInfo hinfo);
     ...
}
```

```
[SpatialPreferPlacementStrategy]
public class Vehicle : Grain, IVehicle {
     IMovingActorMixin movingActor = new MovingActorMixin();
     HazardInfo hazardInfo;
     async Task IMovingActorMixin.Move(Point l_d) {
          await movingActor.Move(l_d);
     }
     async Task<List<ActorInfo>> IMovingActorMixin.FindActors(
    Envelope q) {
          return await movingActor.FindActors(q);
     }
     async Task IMovingActorMixin.StartReactiveSensing(
    Predicates p, Func<ReactionInfo,Task>  reactiveFunc) {
         await movingActor.StartReactiveSensing(p, reactiveFunc);
     }
     Task IMovingActorMixin.StopReactiveSensing() {
          return movingActor.StopReactiveSensing();
     }
     async Task ReactToHazard(ReactionInfo rinfo) {
          // coordinate with other actor that satisfied predicate
          // over this moving actor's fence
          IVehicle other = GrainFactory.GetGrain<IVehicle>(rinfo.
    Id);
          await other.CoordinateHazard(hazardInfo);
     }
     async Task ReactToFireTruck(ReactionInfo rinfo) {...}
```

```
24      Task IVehicle.UpdateHazardInfo(HazardInfo hinfo){...}
25      ...
26  }
```

Listing 3.2: Application-defined cooperative intelligent vehicle(a) top: interface. (b) bottom: actor implementation.

3.4.2 Dolphin's Actor-Based Design

To build Dolphin, we implemented the components of an M-AODB outlined in Section 3.2.3, namely spatiotemporal concurrency control, spatial indexing, and reactive event monitoring using the Orleans virtual actor programming model. Dolphin is a partitioned system where grid partitioning is employed to logically divide the space into cells. Currently, only uniform grid partitioning is supported in the system where developers can define the size of the cell. However, various spatial partitioning methods, such as learned index structures (Kraska *et al.*, 2018; Nathan *et al.*, 2020), can be employed to better deal with data with skewed spatial distribution. In the current work, we focus on the design and implementation of the overall system and defer the exploration of spatial partitioning functions to future work.

Virtual actors (referred to as grains in Orleans[5]) encapsulate data while providing the abstraction of a lightweight thread that executes methods invoked on it sequentially. Thus, concurrent method invocations on the same actor are processed sequentially while concurrent method executions on different actors happen in parallel. Method invocation on virtual actors is exposed through asynchronous function calls (`Promises` in C#), which allows the caller to invoke multiple function calls over multiple actors in parallel.

Figure 3.6 illustrates the mapping of a representative C-ITS application to various components (actors) in Dolphin's architecture. L0 represents a real-world C-ITS application, where the circles around the vehicle icons represent the sensing fences of physical moving objects. A developer models the digital representation of the physical moving objects using *moving actors* shown as a green circle in L1. The fence of a moving actor is shown as green dashed circles in L1. Space is logically partitioned into uniform grid cells.

[5] In the rest of the chapter, we use the terms virtual actor, actor, and grain interchangeably unless stated otherwise.

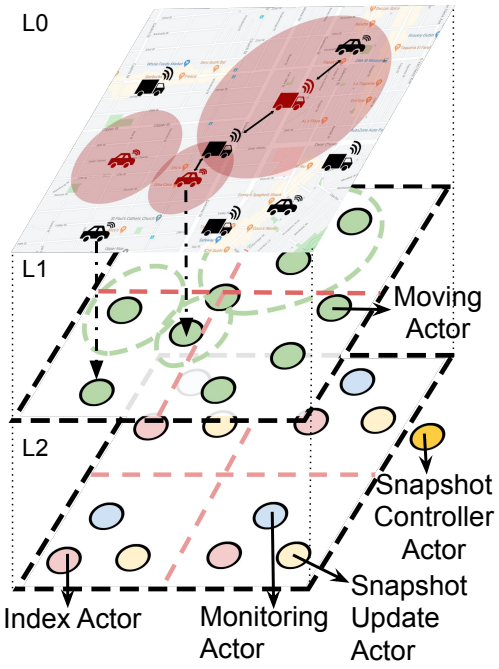


Figure 3.6: Dolphin Design: An Example for C-ITS

L2 shows the functional system components in Dolphin that have been modeled as actors. Each cell consists of an *Indexing Actor* (red circles), *Monitoring Actor* (blue circles), and *Snapshot Update Actor* (light yellow circles). The *Snapshot Controller Actor* (a deep yellow circle in L2) is not present per cell (placed outside the grid) because it is a single actor for the entire application. The roles of these actors have been outlined below.

Indexing Actor. It is responsible for indexing locations of moving actors in the cell. We currently implement R-trees for indexing the location of moving objects in the system.

Monitoring Actor. It is responsible for helping moving actors continuously monitor on their fences against predefined predicates and generate reactions. A monitoring actor acts as an endpoint for communication between moving actors so it receives updates from moving actors in its cell and then relays those updates to all the moving actors who have subscribed to it using the Orleans Streams API.

Snapshot Update Actor. It is only used to support Snapshot semantics. It is responsible for collecting buffered location updates of moving actors in the cell, dispensing received data to related monitoring actors, and distributing them to other snapshot update actors (if necessary), then finally applying the updates by dispatching them to the indexing actor. Since moving actors may move arbitrarily during a snapshot update interval, the data received by a snapshot update actor may contain location data belonging to other cells, which it needs to relay to the appropriate snapshot update actors.

Snapshot Controller Actor. It is only used to support Snapshot semantics. It is responsible for aiding snapshot update actors determine when updates for a snapshot should be reflected on indexes. This coordination is necessary to ensure all the updates for a snapshot have been received from other snapshot update actors before an index is updated in case actors move across cells. It is

also necessary in the case where there are no crossings between cells by actors to ensure snapshots across cells do not diverge and are consistent.

We elaborate upon the workflows for processing the operations exposed by the moving actor abstraction utilizing the aforementioned actors in the next section of this chapter.

3.4.3 Query and Move Workflows in Dolphin

In this section, we present the workflow of `FindActors(`r`)` and `Move(`l_d`)` methods of the moving actor abstraction spanning the functional components (actors) in the system under the Actor-Based Freshness and Snapshot semantics.

Query Workflow under Actor-Based Freshness and Snapshot Semantics

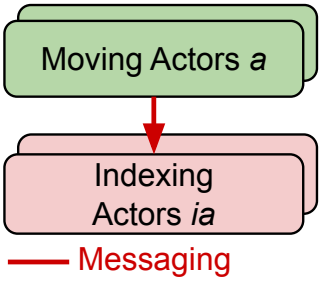
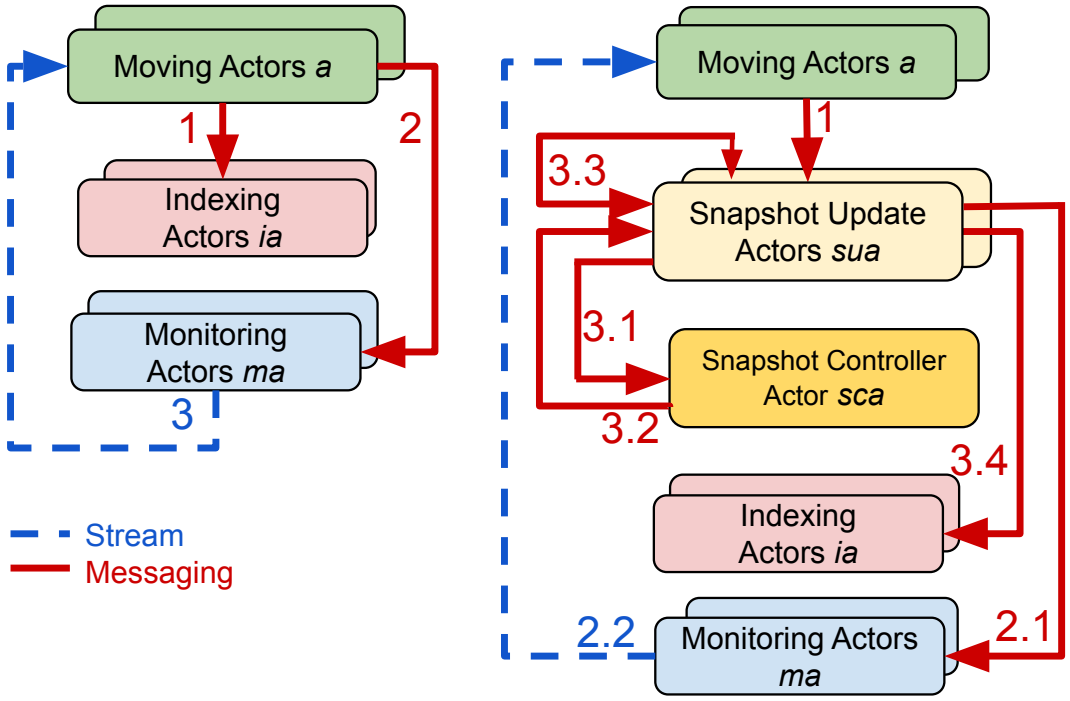


Figure 3.7: Query Workflow

Figure 3.7 shows the query workflow in Dolphin under both Actor-Based Freshness and Snapshot semantics. When a moving actor a receives a `FindActors(`r`)` request, it first determines the cells that are spanned by the requested query range r. Then a asynchronously messages each of the indexing actors ia of those cells to execute an index lookup for the range overlapped by that cell and the requested query range. The communication between the moving actor and the indexing actors is done asynchronously so that the index look-ups are performed in parallel by the indexing actors. Note that in snapshot semantics, every index maintains a monotonically increasing version number that is updated when the index is updated at the end of the snapshot to ensure that a query reads data from indices with the same version number. If the version numbers read from the indexing actors are not the same, then the query result is discarded and the query is retried.

Move Workflow under Actor-Based Freshness Semantics

Figure 3.8 (a) shows the move workflow in Dolphin under Actor-Based Freshness semantics. Since reactions are generated during move, they are explained under the move workflow over the next steps:



(a) Actor-Oriented Freshness semantics (b) Actor-Oriented Snapshot semantics

Figure 3.8: Move and Reaction Workflow in Dolphin

Step 1. When a moving actor a receives a `Move`(l_d) request, it gets the itinerary ($a.iti = \{l_s, l_d\}$ in Definition 2) and finds the cells spanned by $a.iti$. Then a sends the information of this update to the indexing actors in the cells (one in case of a local move, two in case the actor crosses cells) using asynchronous messages. In the meantime, a also updates its state that consists of the location (from l_s to l_d), fence (from f_s to f_d), and subscription to the necessary monitoring actors using the Orleans Stream API so that it can receive updates of other moving actors in cells within range of its fence for generating reactions. Note that only moving actors that have enabled sensing using `StartReactiveSensing`(p,m) subscribe to the stream.

Step 2. Moving actor a sends the update information to the monitoring actors spanned by $a.iti$ so that necessary reactions can be generated on the moving actors that have sensing enabled.

Step 3. When a monitoring actor ma receives updates from moving actors, it publishes the updates on its stream channel[6] that is consumed by the moving actors. Recall from Definition 3, only a moving actor (e.g., a_0) that has enabled

[6]Every cell has a single stream channel with the monitoring actor being the sole publisher and moving actors with sensing enabled being the subscribers.

sensing by invoking `StartReactiveSensing(`p`,`m`)` receives location updates from the monitoring actors on the subscribed stream channel, then checks if the received $a.iti$ satisfies its spatial predicate p against latest fence f (either f_s or f_d, depending on the relation between $a.t_u$ and $a_0.t_u$ shown in Definition 3). If the spatial predicate is satisfied, then the registered method m is executed to perform the reactive action.

Move Workflow under Actor-Based Snapshot Semantics

Figure 3.8 (b) shows the move workflow in Dolphin under Actor-Based Snapshot semantics. Since reactions are also generated during move, they are explained under the move workflow over the next steps:

Step 1. When a moving actor a receives a `Move(`l_d`)` request, it updates its state that consists of its location and fence, and buffers the updated state locally. Aligned with Definition 5, this process creates $a.itin^{S_n}$ for any moving actor and $a_0.AccumulatedFence^{S_n}$ for moving actors who have started sensing. When the timer in the moving actor[7] expires at $S_n.t_i$ mentioned in Definition 4, a finds the cell that the first buffered location refers to, and sends the entire buffered itinerary $a.itin^{S_n}$ to the snapshot update actor sua of this cell. Later data exchanges for snapshot consistency and reaction correctness are carried out between snapshot update actors.

Step 2.1. Every snapshot update actor sua tracks the moving actors that are present in its cell, which is updated at the end of every snapshot. When a snapshot update actor sua receives the buffered location data from all the moving actors that were present in its cell in the last snapshot, it sends the buffered itineraries to monitoring actors ma in the cells that spanned those itineraries.

Step 2.2. When a monitoring actor ma receives the buffered itineraries, it distributes them to the sensing moving actors using the mechanism (Step 3) outlined in the move workflow under Actor-Based Freshness semantics. To fulfill Definition 5, when a sensing moving actor a_0 receives itineraries $a.itin^{S_n}$, it checks them using its predicate p against $a_0.AccumulatedFence^{S_n}$.

Step 3.1. The following steps help switch from a snapshot S_{n-1} to snapshot S_n (see Definition 4). When a snapshot update actor sua receives the buffered

[7] Every moving actor uses a timer configured to the snapshot interval time that fires periodically to signal the start of a snapshot.

data from all the moving actors in its cell in the previous snapshot, it sends a message to the snapshot controller actor sca informing the snapshot update actors they would be sending their buffered location updates to. Recall that there is a single snapshot controller actor in the system.

Step 3.2. The snapshot controller sca tracks cells (active) that have moving actors in it. When the snapshot controller actor sca receives the requests from sua in all active cells from Step 3.1, it responds by informing them about the corresponding suas from whom they should receive messages (final location) from.

Step 3.3. The snapshot update actors sua exchange information between each other to get the needed update data in this snapshot round.

Step 3.4. After sending all data out and receiving all data from the others, every sua sends update information to its cell's indexing actor to update the indices. After all the indices of all the cells are updated, a new snapshot is completed.

3.5 Experimental Evaluation

This section presents experimental evaluation to show that *Dolphin* has met the requirements for reactive moving object data management in Section 3.2.1. Our goals in evaluation are the following:

1. Characterize the relationship between moves and reactions, including aspects such as client-side load and reactive sensing intensity (cf. Sections 3.5.2, 3.5.2 and 3.5.2);

2. Observe how spatial range queries affect Dolphin and their interaction with moves and reactions (cf. Section 3.5.2);

3. Evaluate if Dolphin scales out over multiple machines in a distributed setting (cf. Section 3.5.2);

4. Investigate how the performance of moves and reactions is affected by spatial skew and realistic spatial distributions (cf. Sections 3.5.2 and 3.5.2).

3.5.1 Experimental Setup

Cloud Service and Deployment

We deployed several components for the experiments. The first one was Orleans silo (*Orleans Silo - Microsoft Orleans Documentation* 2020), where virtual actors are activated in a server and run all application logic. We set up our benchmark environment on AWS (*Amazon Web Service* 2020). We employed Amazon DynamoDB (*Amazon DynamoDB - Fast and flexible NoSQL database service for any scale* 2021) for storing the silo membership table (*Cluster Management in Orleans* 2020) and we utilized EC2 (*Amazon EC2 - Secure and resizable compute capacity to support virtually any workload* 2020) for deploying all our instances. Three types of EC2 computing optimized instances were employed. One c5.xlarge instance for controlling and synchronizing benchmark client threads; and one c5.4xlarge instance to simulate moving object client threads. Eight c5.xlarge instances are used as silos to run distributed experiments; and one c5.9xlarge instance to run single-silo experiments. All instances run on Windows Server 2019 Base and Orleans 3.1.2. We placed them in one subnet and one cluster deployment group to reduce network latency (Zou *et al.*, 2011). We built a spatial-preference grain placement strategy (*Grain Placement - Mircosoft Orleans* 2020) to reduce the number of messages across silos (cf. Subsection 3.5.2).

Query, Move, and Reactions

Our experiments start by initializing a set of moving actors. All clients issue a sequence of requests as soon as possible. Client requests only include queries and moves to moving actors. We define the following processes in Dolphin:

Query. Queries are range queries using the method `FindActors(r)`. In our experiment, we fix the query window size for simplicity.

Move. Moves are `Move(`l_d`)` requests. Moves may trigger a reactive method of all moving objects that have started reactive sensing accordingly. Client benchmarks were responsible for interpreting movement models and generating the necessary move calls.

Reaction. A fraction (0-100%) of moving actors are set to call `StartReactiveSensing(`p`,` m`)` at initialization so that each triggers reactive action m when other actors' move requests satisfy p against its fence. In our

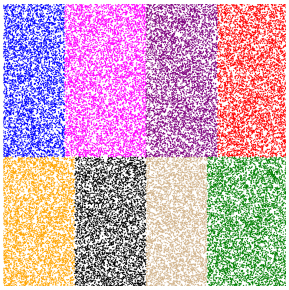


Figure 3.9: uniform distribution benchmark dataset.

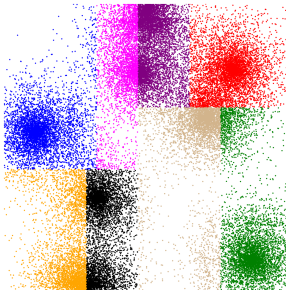


Figure 3.10: gaussian distribution benchmark dataset.

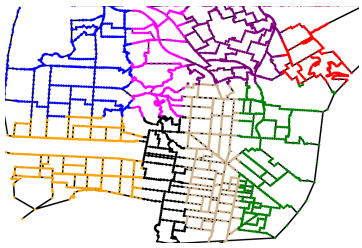


Figure 3.11: c-its scenario benchmark dataset.

experiments, for all moving actors, we chose *cross* as a spatial predicate. We also defined the reactive method as building a connection with the actor who crosses the fence.

This section analyses performance issues concerning underlying workflows associated with *queries* and *move*. We called them client requests as they are triggered by clients. Differently, reactions themselves occur when reactive sensing is on. Related work on moving objects considers two kinds of requests: queries and move, but without reactions. Our work better reflects the real world as we include reactive behaviors; as such, our experiments cannot be directly comparable to experiments performed in related work. On the other hand, reactions come with a processing cost. Therefore, while in other systems, the *move* request does not imply additional processing, moves in Dolphin are more expensive than in other works due to associated reaction processing.

Benchmark Implementation

We conducted experiments under three benchmarks, namely uniform distribution benchmark (cf. Figure 3.9), Gaussian distribution benchmark (cf. Figure 3.10) and C-ITS scenario benchmark (cf. Figure 3.11) . In the first two benchmarks, moving objects can move freely in a 2D space under two distributions – uniform or Gaussian. In the C-ITS scenario benchmark, objects are constrained to move along a real-life urban network. We set the configuration of our benchmarks from (Chen *et al.*, 2008; Sowell *et al.*, 2013), adapting their settings to our scenario (cf. details at Subsection 3.5.1).

Uniform Distribution Benchmark. In the uniformly distributed dataset, moving actors are instantiated uniformly at random in positions in a fixed size square space and move towards random directions with a random speed under a speed limitation.

Gaussian Distribution Benchmark. Gaussian distributed dataset models a skewed spatial workload. In this configuration, moving actors are clustered around a set of hotspots. Those hotspots are uniformly initialized at random locations, and moving actors are distributed around each hotspot based on a Gaussian distribution. Actors move at a speed related to the location of the hotspot. A moving actor moves faster when it is farther away from hotspots and takes a slower movement when close to hotspots (Chen *et al.*, 2008; Sowell *et al.*, 2013).

For both uniformly and Gaussian distributed datasets, if a moving actor moves across the borders of the square space, it is bounced back to the opposite direction and moves the same excess distance that should not exceed the border. If this is not possible, the moving actor remains stationary and expects a new update.

C-ITS Scenario Benchmark. C-ITS scenario benchmark models constrained spatial movement with the aim of observing the performance of Dolphin in a realistic setting. For this benchmark, we used the TIGER/Line data (Bureau, 2020) corresponding to “All roads2019 San Francisco Country". Moving actors behave as digital twins for vehicles in San Francisco. According to the authors of (Zhou, 2020), we initialized 38000 vehicles[8] and limited the space to the main part of San Francisco city. Furthermore, vehicles move along the road network at a fixed speed (22m/s (Wikipedia, 2020)), while following two rules: (1) A moving actor keeps moving along an edge; and (2) when the moving actor arrives at the conjunction of edges, it chooses to follow a random direction along the edges and continues.

Workloads

Table 3.1 presents our workload settings, where parameters are adapted whenever possible from previous benchmarks (Sowell *et al.*, 2013; Chen *et al.*, 2008). We deviate from previous settings in the following: we set a maximum speed of moving actors to be 80km/h, which is the speed limit for urban areas in many countries (Wikipedia, 2020). We adopted the proportion between moving object numbers and space sizes from the study in (Sowell *et al.*, 2013). In contrast to previous studies that focus on notions of virtual time, *e.g.*, ticks

[8] 5700 transportation network companies (TNCs) vehicles operate during peak period, and those trips represent 15% of all intra-San Francisco vehicle trips. For the sake of simplicity, we estimate 5700/15%=38000, which may be less than real traffic because more commute trips exist in peak period.

Table 3.1: Workloads for Benchmarks.

Parameters	**Benchmarks**		
	Uniform	**Gaussian**	**C-ITS**
Number of Servers	1, 2, 4, 8	8	8
Client Threads per Server	8 (single server) or 4 (multiple servers)	4	4
Number of Moving Actors	5000 per server	40000	38000
Space Size (km^2)	100,199.94,400,799.98	799.98	154
Number of Cells	100,196,400,784	784	805
Number of Hotspots	/	8,80,400,800,8000,40000	/
Fence Size (m^2)	1000×1000		
Query Size (m^2)	1000×1000		
Max Speed (km/h)	80		
Reactive Sensing Percentage (%)	0,**12.5**,25,50,100	12.5	12.5
Query Ratio (%)	**0**,20,40,60,80,100	0	0
Snapshot Interval Time (s)	1 or 4		

or timestamps (Chen *et al.*, 2008; Sowell *et al.*, 2013), our study configured the base number of moving actors for one silo to 5,000. We aimed to match a workload that could be easily handled by a silo while delivering reasonable real-time performance.

In line with previous studies, however, the fence and query sizes were chosen based on a realistic setting, where vehicles are interested in a range of 1km. We set the default query rate to be 0, which means all client's requests are move requests to maximize the reaction influence. There is no guide for reactive sensing percentage setting in previous studies. We then set and assume it to be 12.5% based on a reality assumption. We set the number of cells in the spatial partitioning scheme used (100) as well as the snapshot interval time (1 sec) based on corresponding tuning experiments. Based on snapshot interval time tuning experiments, for a single server, the system can achieve snapshot update within 1 sec. Also, the system provides a reasonable reaction throughput and 1 sec can satisfy most use cases. The snapshot interval time for distributed experiments are set to be both 1 sec and 4 secs.

In our setup, c5.xlarge instances were used for the Orleans silo in most experiments except a experiment shows in Section 3.5.2. Every client thread was configured to send requests as fast as possible to moving objects so that we can increase update frequency and thus trigger reactions more quickly to the application. Our focus was on the scalability of updates and reactions, which stands in contrast to previous studies, where each moving object only sends updates infrequently. In previous studies, scalability is sought in terms of the numbers of moving objects (Chen *et al.*, 2008). Based on our workload tuning experiments results (cf. Figure 3.12), clients only send move requests because we emphasized exploring reaction behavior. When client threads are

8, the workload saturates our 4vCPUs server. Meanwhile, moving actors update frequency is around every 0.6 secs and every 0.5 secs under Actor-Based Freshness and Actor-Based Snapshot semantics, respectively. For distributed experiments, we choose not to saturate each server to give some resources to extra overhead. In this sense, we used four client threads for each server.

3.5.2 Experimental Results

This section presents the experimental results. For brevity, we use Fresh and Snap to express 'Actor-Based Freshness semantics' and 'Actor-Based Snapshot semantics', respectively.

How do reactions behave as Dolphin is faced with increasing move workloads?

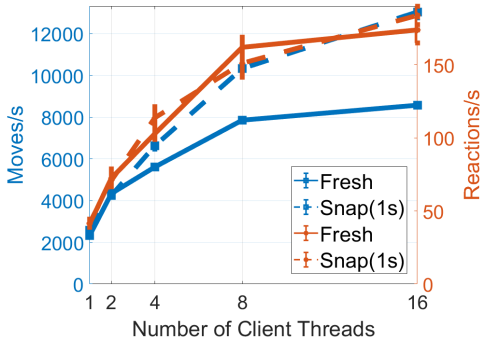


Figure 3.12: Correlation of reaction and move throughputs under increasing client-side workload intensity.

In this experiment, we increased the client-generated move workload up to the saturation point of a single server, and observed the effects on reactions. Figure 3.12 shows that reaction throughput increases along with move throughput for both Fresh and Snap(1s). This is expected because more moves increase the chances to trigger more reactions. However, reaction generation also increases the resource consumption on the server side, which limits the attainable throughput of moves. Since Snap(1s) batches reaction generation at the snapshot interval of 1s, its move throughput is more resilient to interference and is 1.52x higher than Fresh at 16 client threads. We found that 8 client-thread workloads saturate the system well. With this setting, each moving actor is able to report a move on average every 0.637 sec and 0.484 sec for Fresh and Snap(1s), respectively, with corresponding move request latencies of 1.02 ms and 0.77 ms measured at the client side. This shows that our system can handle most cases in real life.

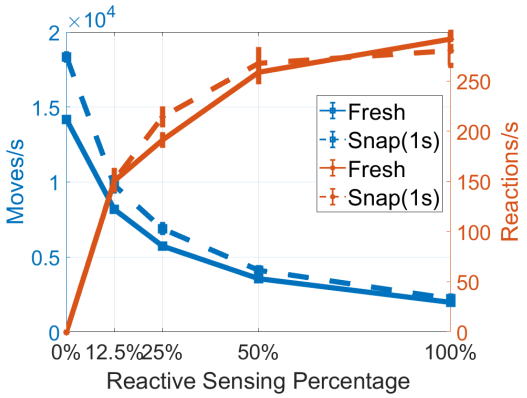


Figure 3.13: Inverse relationship between reaction and move throughputs under excessive reactive sensing.

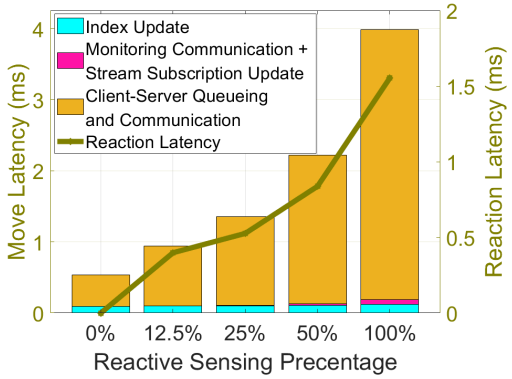


Figure 3.14: Move latency breakdown and reaction latency for excessive reactive sensing in Actor-Based Freshness semantics.

How do reactions behave under a saturated move workload and excessive reactive sensing?

In this experiment, we fixed the workload generation at the saturation level of 8 client threads, and increased the reactive sensing percentage. Since the system was already saturated with the default 12.5% reactive sensing percentage, we aimed at observing if Dolphin's performance degrades gracefully with excessive reactive sensing.

Figure 3.13 shows that move throughput goes down as expected, but counter-intuitively, reaction throughput increases. The reasons for that are: firstly, with increasing percentage of moving actors doing reactive sensing, more reactive methods are issued by a single move; moreover, move throughput going down makes every moving actor update less frequently, which results in moving actors becoming prone to move a longer distance. Longer distances also increase the possibility of moves crossing spatial fences. For decreased move throughput, we believe that since reactions are more expensive, their processing queues up move requests.

To validate this effect, we break down move latency and compare it with reaction latency under Fresh (cf. Figure 3.14). We confirm that move latency is mostly caused by client-server queuing and communication, while server-side components such as index update together with monitoring communication plus stream subscription update represent a small fraction of the latency. We also observe that reactions are more expensive than moves: Reaction latency at the server side is generally higher than move latency, *e.g.*, 0.526 ms vs.

0.144 ms at 25% reactive sensing percentage. However, we remark a dilation of both move and reaction latencies due to overload at higher reactive sensing percentages. The latter effect reduces the number of moves processed by the system, which in turn also limits the system's potential to generate more reactions. This interplay explains why reaction throughput tends to flatten above 50% reactive sensing in Figure 3.13. We observed similar results in Snap.

Can Dolphin leverage additional server capacity to improve reaction and move throughtputs under a high reactive sensing percentage?

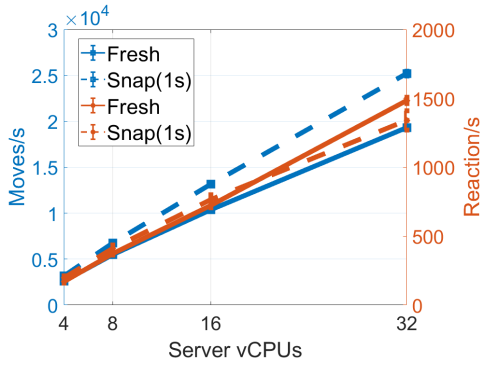


Figure 3.15: Reaction and move throughputs with more server vCPUs and 50% reactive sensing.

We now understand that when system capacity is insufficient to handle an excessive reactive sensing situation, the performance of Dolphin degrades. In this experiment, we fix the reactive sensing percentage of moving actors to 50%, which is an excessive reactive sensing situation, and increase the server capacity.In order to support that, c5.x9large instance with 36 vCPUs is used for Orleans silo in this experiment. Meanwhile, workloads were controlled to saturate server capacity while maintaining latency within an acceptable varied range. By doing that, we aimed to understand if increasing additional server capability helps to handle the excessive reactive sensing situation.

In Figure 3.15, move reaction throughput increased linearly along with the increase of server capability for both Fresh and Snap(1s). Move throughout increased 7.44x under Fresh and 7.91x under Snap(1s) when server capacity increased 8x. Snap(1s)'s throughput is more resilient to interference because it batches requests. In sum, when the server hits the excessive reactive sensing percentage situation in which queuing of tasks degrades system performance, the issue can be solved by allocating sufficient server capacity.

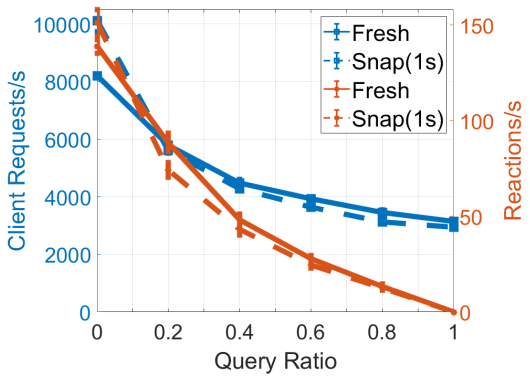


Figure 3.16: Client request and reaction throughputs under increasing query ratio.

Figure 3.17: Latency breakdown with increasing query ratio under Freshness semantics.

How does increasing spatial range query ratio affect Dolphin's request and reaction throughputs and latencies?

Except for the new reactive feature, Dolphin provides range query and moves like other spatial data management systems. We have explored how reactions and moves affect the system. We now proceed to describe the performance of Dolphin regarding different kinds of client requests. This scenario is a mix of "query plus move" in different query ratios in order to understand trade-offs in workload configuration and overall performance costs. Query ratio is the percentage of client query requests against all client requests. We varied query ratio from 0% to 100%, which means varied client requests from composed by 100% move, 0% query to 100% query, 0% move. Meanwhile, 12.5% moving actors are reactive moving actors in this experiment to understand how query ratio affects the reaction generation.

Figure 3.16 shows that for both Fresh and Snap(1s), as expected, reaction throughput decreased with increased query ratio because reactions are generated by move. Client request throughput also goes down along with increased query ratio. This effect can be analyzed by the interplay of query and reaction expenses and a breakdown of the client request latency under Fresh. Our results in the latency analysis of Figure 3.17 show that a query is about 3.44x-4.10x more expensive than a move in this scenario. However, client request throughput only degrades by 2.60x. That is explained by reactions being more relatively expensive in Dolphin. With the decrease of moves, the influence of reactions decreased. Therefore, with the increase of query ratio, the degradation in client task throughput is not so dramatic in Dolphin. A similar result is found in Snap(1s) as well.

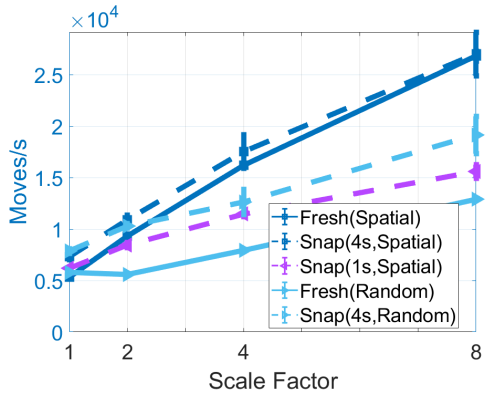


Figure 3.18: Move throughput scalability with increasing servers and workloads.

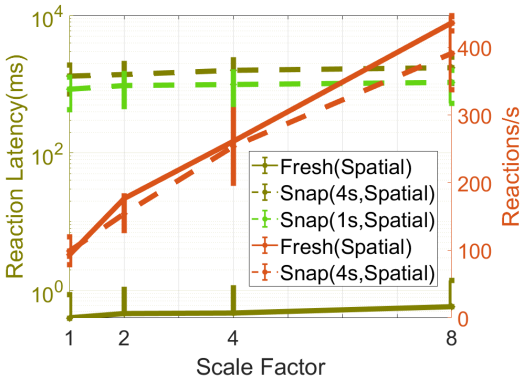


Figure 3.19: Reaction latency and throughput with increasing servers and workloads.

Can Dolphin scale out move and reaction throughputs with increasing servers and workloads?

To test the scalability of Dolphin, we scale the number of servers, datasets, and workloads (cf. Table 3.1). All client requests are move requests. In Orleans, the actor runtime decides which server to activate an actor (grain) on, which is called grain placement. The default grain placement in Orleans is to activate grains on a random server in the cluster (*Grain Placement - Mircosoft Orleans* 2020). In Figure 3.18, move throughput Fresh(Random), Snap(4s, Random) increases non-linearly afterward because communications among servers increase with the number of servers. To solve that, we developed a customized spatial grain placement that partitioned the whole space by employing a KD-tree (Orenstein, 1982). Spatially close moving actors having a higher chance to communicate with each other can then be placed on a server.

Move throughput scalability under spatial grain placement, especially of Fresh(Spatial), improved substantially. However, throughput is still not perfectly linearly increased because communication between servers cannot be eliminated, and moving actors may move outside the placement partitions along with time. Moreover, due to the single component to control snapshot update, the throughput of Snap(1s, Spatial) is still not optimal. That can be solved by enlarging the snapshot interval time to reduce the overhead of the snapshot update, seen from the move throughput Snap(4s, Spatial). We employ our customized spatial grain placement for the following experiments and omit that in legends.

In Figure 3.19, as the system scales, reaction throughput Fresh(Spatial) and Snap(4s, Spatial) increase along the same trend as the corresponding move throughput in Figure 3.18. For Snap variants, reaction latency is constrained by snapshot interval time because reactions are only triggered during the snapshot update. For two different snapshot interval times, 50th-percentile reaction latency from 1 sec interval time is 1.53x-1.73x faster than from 4 secs. But they are both uncompetitive to the reaction latency of Fresh, which provides a more than one thousand times faster reaction. So for applications extremely sensitive to real-time reactions, Actor-Based Freshness semantics is recommended.

How do move and reaction throughputs behave under increasing spatial skew?

We employ a Gaussian distributed workload (cf. Table 3.1) to answer this question. All client requests are move requests. We analyzed the move and reaction throughput behavior with increasing spatial skew in data. In Figure 3.20, as expected, move throughput goes down with more spatial skew, because skewed data increases the workload to a single cell, which causes more queuing and communication in those over-loaded cells.

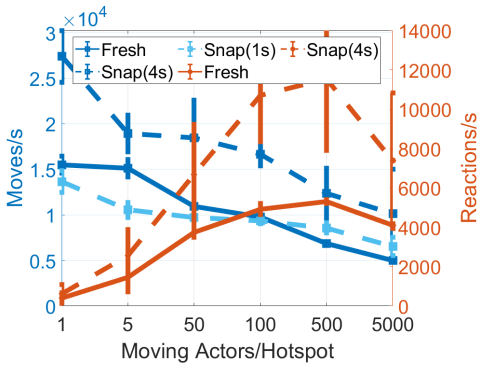


Figure 3.20: Move and reaction throughputs with increasing spatial skew.

We can also see that Snap(4s) provides a better move throughput performance compared with Snap(1s). For both Fresh and Snap(4s), reaction throughput increased from near-uniform data (1 moving actor/hotspot) to 500 moving actors/hotspot data, but then decreased for further skew. The reason for increasing reaction throughput is that due to the fixed number of moving actors, with fewer hotspots, more moving actors gather together around one hotspot. In this sense, one single move trajectory is more likely to satisfy predicates against more fences. However, reactions should also decrease along with the decrease of move throughput as shown in Subsection 3.5.2. Also, in our Gaussian movement setting, a moving actor in more skewed data would have a lower chance to move faster, which would

Table 3.2: Results for C-ITS scenario benchmark.

Results	**Semantics**		
	`Fresh`	`Snap(1s)`	`Snap(4s)`
Moves/s	3349.99(±34.65)	5211.61(±2583.67)	9276.87(±2583.67)
50% Latency (ms)	6.26	0.59	0.55
99% Latency (ms)	53.55	89.94	60.84
Reactions/s	1925.86(±49.31)	120.29(±32.54)	287.86(±134.29)
50% Latency (ms)	22.56	5605.49	8261.29
99% Latency (ms)	1092.66	18835.36	22364.91

decrease the length of a trajectory. Those counterfactors make the reaction throughput flatten and finally decrease as in Figure 3.20.

How does Dolphin perform in the C-ITS scenario benchmark?

We generate workload according to the C-ITS scenario (cf. Table 3.1) to understand how Dolphin behaves in a realistic benchmark. Table 3.2 shows the results. We observe that the 50% latency and 99% latency of vehicle move under Fresh is 6.26 ms and 53.55 ms, respectively. The corresponding latencies under Snap(1s) is 0.59 ms and 89.94 ms, respectively; and 0.55 ms and 60.84 ms, respectively, for Snap(4s). According to ETSI ITS Specifications 2020 (*ETSI EN 302 890-2 V2.1.1 (2020-10)* 2020), the threshold for waiting to be serviced by a GNSS Positioning Correction service of a Roadside ITS is 2 seconds. So the move latency provided by Dolphin in the C-ITS scenario benchmark satisfies this requirement well. Also, we note that the 50% reaction latency of Fresh is 22.56 ms, which provides a near-real-time reaction latency. Snap(4s) provides faster reaction and higher move and reaction throughput than Snap(1s). Therefore, 4 secs is a more suitable snapshot interval time for our C-ITS scenario benchmark.

3.6 Related Work and Discussion

As modern interactive and data-intensive applications demand a scalable and elastic application tier (Bernstein *et al.*, 2014), actor programming frameworks such as Akka (*Akka - Build Powerful Reactive, Concurrent, and Distributed Applications More Easily* 2020), Erlang (*Erlang-Build massively scalable soft real-time systems* 2020), Orbit (*Orbit* 2020), and Orleans (*Orleans Microsoft - Microsoft Orleans Documentation* 2021) are becoming popular implementation options that have led to significantly increased programmer productivity. In particular, the abstraction of virtual actors (Bykov *et al.*, 2011) in Orleans

considers actors as modular and stateful virtual entities in perpetual existence, facilitating a one-to-one mapping to moving objects, among other IoT scenarios. The Orleans runtime increases productivity and convenience for developers by handling failure, automatically and dynamically managing actors' life cycle and balancing load across servers, which also helps developers achieve the desired scalability and availability (Bernstein, 2018; Bernstein *et al.*, 2014). Orleans has been used in many production services, which validate its readiness and reliability.

By building upon virtual actors and Orleans, Dolphin leverages many of its characteristics. However, scalability for reactive moving actor applications does not come by default. Specifically, to provide scalability and reliability, physical instantiations of actors in Orleans are distributed across the silo instances (Bernstein *et al.*, 2014). Actors are activated on silos based on the calculation of an actor placement strategy. However, Newell *et al.* (2016) pointed out, Orleans built-in actor placement policies are insufficient to achieve scalability, which can also be seen in our experiments, particularly in Figure 3.18. For instance, the default RandomPlacement strategy results in good load balancing, but this strategy ignores spatial locality in interactions among moving actors. The latter leads to unnecessary latency and overheads, which significantly degrades system scalability. These effects are avoided in Dolphin by our spatially aware system-level actor design. Moreover, existing actor programming models, including Orleans, lack built-in support – with well-defined concurrency semantics – for the reactive data management functionalities required by reactive moving object applications, complicating the development of these applications.

Previous investigations about location-aware sensors, devices, and infrastructures have studied how to represent, manage, and query moving objects (*e.g.*, Guting and Schneider (2005) and Güting *et al.* (2006)), as well as how to define and implement correct behaviors of moving object databases (*e.g.*, Lu and Güting (2014) and Wolfson *et al.* (1998)). Such studies on moving object databases are mainly focused on data structures, algorithms, and architectures for supporting past, present, and near-future queries (Pelanis *et al.*, 2006; Sidlauskas *et al.*, 2014; Sowell *et al.*, 2013; Jensen *et al.*, 2004; Jensen and Pakalnis, 2007). However, reactive moving object applications require novel reactive API as demonstrated by our Moving Actor abstraction, which is not supported in existing moving object databases and libraries.

RxSpatial (Hendawi *et al.*, 2017) is probably one of the only works toward reactive spatial data management. RxSpatial married the Microsoft SQL Server Spatial Library (Andzic *et al.*, 2020) with ReactiveX (*Microsoft Reactive Framework - ReactiveX: An API for asynchronous programming with observable streams* 2020) to implement a real-time reactive spatial library. RxSpatial provides a reactive spatial API, employs a spatial index structure for moving object queries, and continuously monitors and detects the intersection and distance between moving objects (Shi *et al.*, 2016a; Shi *et al.*, 2016b; Hendawi *et al.*, 2016). However, the reactivity model in RxSpatial is limiting in terms of supporting reactive moving object applications. Reactive moving objects in RxSpatial need to explicitly subscribe to other objects in order to monitor the relation between them, which is not an efficient solution. For example, in order to implement the the reactive sensing API in our platform, every reactive moving object would have to subscribe to the movements of all the other moving objects in the system, which could be highly expensive. Dolphin, by contrast, allows reactive moving objects to subscribe to intensional representations in terms of fences and spatial predicates, obviating explicit object-to-object subscriptions and enabling a partitioned actor-based design.

There are also some solutions that provide spatial event monitoring by using stream processing systems. The studies conducted in (Ali *et al.*, 2010a; Miller *et al.*, 2011; Kazemitabar *et al.*, 2010; Ali *et al.*, 2010b) support continuous spatial queries for monitoring. Galic *et al.* (2017) proposed a distributed spatio-temporal mobility data stream framework to support continuous queries over streams. GeoFlink (Shaikh *et al.*, 2020) extended Apache Flink to support spatial data types, indexes, and continuous queries over spatial data streams. The stream processing abstraction in these systems, often in the form of a topology of stream operators, is suitable for specifying a set of bulk operations over a large number of data items, such as transformation, filtering, join, and aggregate. However, it is unnatural and difficult to use this abstraction to specify complex application logic, such as heterogeneous behaviors of reactive moving objects. The latter creates an impedance for practitioners to implement the application tier of reactive applications. Additionally, stream processing systems do not naturally fulfill the design objective O1 in Section 3.2.1, which introduces complexities for developers in plugging together the concurrency semantics of these engines with those of spatially aware data stores. By contrast, Dolphin enriches the popular actor programming model with features of reactive moving object data management as well as with principled actor-based

spatiotemporal concurrency semantics covering movement, spatial queries, and reactions.

3.7 Conclusion

Emerging moving object applications require additional support for reactivity based on spatial sensing of dynamic data. Furthermore, these applications require scalability, elasticity and low-latency, all of which affect programmer productivity. We proposed a scalable, elastic, reactive, spatial data caching layer for moving object applications to meet these requirements holistically. We posited a new data management system architecture for reactive moving object applications by adding reactive moving object data management functionality in actor-oriented databases. We judiciously designed a new programming model for this new class of moving actor-oriented databases and carefully implemented it in a system called Dolphin, which extends the Microsoft Orleans virtual actor runtime. Our experiments using both micro-benchmarks and a C-ITS scenario benchmark showed that Dolphin meets the real-time performance requirements for spatial data ingestion, querying, and reactivity. Moreover, our experiments highlighted that our system seamlessly scales out across multiple machines.

We hope that Dolphin, for which we plan to release artifacts upon acceptance, can help alleviate the complexity of development and data management for reactive moving object applications. Additionally, given the huge potential for societal impact of these mobile Internet-of-Things applications, we aspire that our work may serve as a catalyst to further motivate the research community to explore the possibilities and challenges of enriching actor runtimes with programming abstractions and algorithms tailored to this scenario.

4 An Evaluation of Spatial Partitioning Techniques to Handle Skew in Moving Actor-Oriented Databases

Reactive moving object applications are emerging in domains such as cooperative intelligent transportation systems, collaborative robotics, and location-aware trans-reality games, among other mobile Internet-of-Things scenarios. These applications necessitate tracking of the movement of populations of moving objects – e.g., vehicles on the road – and low-latency reactions to situational conditions – e.g., a vehicle becoming aware of a hazard and coordinating a response with other vehicles in its vicinity. Moving Actor-Oriented Databases (M-AODBs) have recently been introduced as a novel data management architecture for such reactive moving object applications. However, tolerating spatial skew in the distribution of reactive moving objects remains a significant challenge for M-AODBs supporting these applications, given the impact of skew on tail latencies and load balance. To achieve scalability through parallelism and distribution, M-AODBs utilize spatial partitioning, which directly affects load balance in the split of movement processing as well as tail latencies in the communication induced by reactions.

In this chapter, we conduct a comprehensive evaluation on the impact of different spatial partitioning techniques on the performance of a state-of-the-art M-AODB, called Dolphin, under various skewed spatial distributions. Our experiments reveal how partitioning techniques behave in the light of significant parameters in Dolphin such as varied partition capacity, reactive sensing rate, reactive range, and query rate. Furthermore, we observe how different partitioning techniques in Dolphin perform under skew when scaling across servers. Results are obtained that reveal how to choose the most adequate partitioning techniques among the ones evaluated under different scenarios. Moreover, the results include a breakdown of the most significant factors affecting system performance, which comprise primarily the communication

dissemination overhead in reaction generation, but also the coefficient of move load balance among partitions.

4.1 Introduction

Reactive moving object applications are increasingly attracting attention from both industry and academic research related to the mobile Internet-of-Things (Shi *et al.*, 2016a; Costa, 2018) in areas spanning cooperative intelligent transportation systems (Mitsakis *et al.*, 2020; Bussche, 2020; Nguyen *et al.*, 2020), collaborative robotics (Duckett *et al.*, 2018; Albani *et al.*, 2017; McLellan, 2020), location-aware trans-reality games (Gutierrez *et al.*, 2012; Lindley and Eladhari, 2005), and emerging criminal trajectory tracking systems (Abd El-Aziz *et al.*, 2012; Elrefaei *et al.*, 2017; Tundis *et al.*, 2020), among others. Reactive moving object applications both track the movement of objects in space as well as allow objects to trigger reactions based on the movement of other objects. Critically, these applications need to scale to large populations of moving objects and process reactions with low latency.

In Chapter 3, we introduced a novel approach to providing data management system support to reactive moving object applications, namely that of Moving Actor-Oriented Databases (M-AODBs). An M-AODB is a scalable and distributed reactive actor-oriented database exposing the novel abstraction of moving actors. Moving actors allow applications to specify object movement updates, query the current locations of moving objects, as well as process reactions to changes in object locations. In addition, moving actors facilitate object-based application modeling under simple concurrency semantics for reactions, avoiding the scalability and complex execution semantics issues in trigger systems (Widom and Ceri, 1996; Hanson *et al.*, 1999).

To achieve scalability through parallelism and distribution, M-AODBs utilize spatial partitioning to allocate reactive moving objects to spatial regions. Processing of movement is then split across these regions, while the generation of a reaction from movement may affect several nearby regions. Therefore, spatial partitioning directly affects load balance in the split of movement processing as well as tail latencies in the communication induced by reactions.

As such, a significant problem in scaling reactive moving object applications with M-AODBs is that of handling skew in the spatial distribution of moving

objects. For instance, in collaborative intelligent transportation systems, vehicles move along the road network and are thus not uniformly distributed in space. Furthermore, vehicle density in hotspots (e.g., city centre) is higher than in other areas (e.g., a rural area). Spatial skew can lead to load imbalance if certain regions get overloaded with moving objects; additionally, spatial skew can lead to high tail latencies if significant inter-region communication is induced by reactions.

We observe that spatial skew patterns in reactive moving object applications are fairly stable, since they reflect the concentrations of objects around areas of interest (e.g, city centre vs. rural area). Thus, the spatial partitioning utilized by the M-AODB could exploit knowledge about the expected spatial distribution of reactive moving objects.[1] Fortunately, there is a great variety of spatial access methods (Gaede and Günther, 1998), offering a wide range of trade-offs, that can be utilized to realize spatial partitioning in a manner that is sensitive to skew. However, the impact of these techniques on reactive moving object data management – and M-AODBs in particular – is unknown.

In this chapter, we investigate the impact of a set of classic spatial partitioning techniques on the performance of Dolphin, a state-of-the-art M-AODB, under a variety of skewed spatial distributions. To allow for scalable movement processing and low-latency reactions, we discuss constraints that a spatial partitioning method must satisfy to be integrated with Dolphin, and how existing methods can be adapted to allow for partitioning both within and across multiple servers. Additionally, to reflect realistic moving object scenarios, we focus on workloads proposed in previous benchmark studies (Chen *et al.*, 2008; Sowell *et al.*, 2013), but adapted to our scenario discussed in Chapter 3, comprising mainly variations of Gaussian skew. We compare the results of five different spatial partitioning methods to observe how they influence the performance of movement and reaction processing in Dolphin. Specifically, we delve into details about factors that affect parallelism and communication intensity to explain the impact of each partitioning method in the chosen M-AODB.

In summary, our contributions in this chapter are as follows:

[1] We note that spatial skew could change over time or due to seasonal patterns, e.g., the city centre may exhibit a high concentration of objects during the day, but low at night. In such cases, periodic re-partitioning techniques could be employed (Taft, 2017). We leave the exploration of this possibility, however, to future work.

1. We discuss how to integrate spatial partitioning techniques into a state-of-the-art M-AODB, Dolphin, including how to adapt five classic spatial partitioning techniques to work within and across servers.

2. We conduct a thorough evaluation of the spatial partitioning methods in Dolphin based on a synthetic Gaussian distribution benchmark with controllable spatial skew. The evaluation delves into explanatory metrics related to load balance and communication, revealing how different partitioning techniques behave under varied use scenarios.

The remainder of this chapter is organized as follows. In Section 4.2, we review movement, query, and reaction processing in M-AODBs, particularly Dolphin, as well as design constraints and goals for spatial partitioning in these systems. Section 4.3 presents the spatial partitioning methods that we chose to evaluate in this study, along with how they were adapted to work in Dolphin. Section 4.4 presents our experimental setup and results, including discussion of the main effects observed. Related work is reviewed in Section 4.5, while Section 4.6 concludes the chapter.

4.2 Moving Actor-Oriented Databases

Moving actor-oriented databases (M-AODBs) are actor-oriented databases that support spatial queries, movement, and associated reactions through the abstraction of moving actors. The goal of M-AODBs is to provide scalability for reactive moving object applications while facilitating actor-oriented application modeling and construction. In M-AODBs, moving actors can be conceptualized as digital twins (Cimino *et al.*, 2019) of reactive moving objects. As such, moving actors are the construct for management of data originating from reactive moving objects in a cloud-based server infrastructure.[2]

Diverse architectural choices can be made to design an M-AODB implementing the abstraction of moving actors. For concreteness, we focus our attention in this study on a state-of-the-art M-AODB, Dolphin, introduced in previous Chapter 3. This section first reviews how Dolphin supports spatial range queries, movement, and associated reactions (Section 4.2.1). Subsequently,

[2]While in practical scenarios potentially remote physical moving objects need to exchange data with such a cloud-based infrastructure, we do not include this communication in our study, focusing on scalability and reactivity of data management with moving actors instead.

design goals for spatial partitioning methods are set forth, building the basis for our choice and integration of these methods in Dolphin (Section 4.2.2).

4.2.1 Partitioned Spatial Data Processing in M-AODBs

Moving actors offer the API and core application programming construct for M-AODBs. Internally, however, M-AODBs are architected around a set of components that partition, index, and monitor events on moving actor data. In particular, to achieve scalability through parallelism and distribution, M-AODBs utilize spatial partitioning by allocating reactive moving objects to spatial regions, which we term *logical cells* in the remainder of this chapter. Within logical cells, spatial indexing and event monitoring components further support the core spatial range query, movement, and associated reaction processing in an M-AODB.

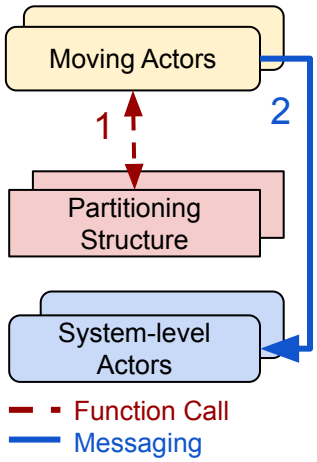


Figure 4.1: Moving actors, partitioning, and system-level actors in Dolphin.

Dolphin represents the internal components associated to each logical cell as a set of *system-level actors*. The relationship between moving actors, spatial partitioning, and system-level actors is depicted schematically in Figure 4.1. Applications specialize moving actors to represent reactive moving objects. As the application interacts with the moving actor APIs, moving actors in Dolphin will engage system-level actors for data management tasks. This is mediated by a partitioning structure, which allows moving actors to address the system-level actors associated to the logical cells affected by any particular task. For efficiency, the partitioning structure is accessed directly through function calls by moving actors and initialized with Dolphin at startup. Communication between actors is achieved by leveraging the abstractions and services of an AODB. In the case of Dolphin, we build on the AODB substrate of Microsoft Orleans (Bernstein *et al.*, 2017b), and thus all actors are virtual actors interacting via asynchronous RPCs (Bernstein *et al.*, 2014; Bernstein and Bykov, 2016).

Dolphin provides two application-level concurrency semantics, namely Actor-Based Freshness semantics and Actor-Based Snapshot semantics, to accommodate different application use scenarios and requirements. Actor-Based Freshness semantics provides the most recent results in M-AODBs, but cannot guarantee these results are observed at a consistent point in time. By contrast, Actor-Based Snapshot semantics provides point-in-time images of M-AODBs, but reaction latency is distributed around the snapshot update time. Given the cost of taking a distributed snapshot, the latter implies that scenarios demanding low latency reactions, e.g., on the order of milliseconds, should privilege Actor-Based Freshness semantics; meanwhile, scenarios that can accept larger latencies for reactions, e.g., on the order of seconds, but demand higher data consistency should opt for Actor-Based Snapshot semantics.

Data processing between moving actors, spatial partitioning structures, and system-level actors differ between these two semantics, but the partition lookup mechanism is the same. To focus on exploring the impact of partitioning techniques on reaction latency, we only discuss Actor-Based Freshness semantics in the remainder of the chapter. We have validated that the trends we observe for the latter are similar for Actor-Based Snapshot semantics, albeit variance in request throughput and reaction latencies is higher overall due to the snapshotting mechanism.

At a high level, Dolphin utilizes spatial indexing inside logical cells to speed up processing of spatial range queries as well as Orleans streams to mediate the triggering of reactions from movement events. These two system-level functionalities are encapsulated in two different actor types, indexing actors and monitoring actors, where one instance of each type is associated to each logical cell. We review the approaches taken in Dolphin for spatial range query, movement, and reaction processing in more detail in the following.

Spatial Range Query Processing

The moving actor abstraction exposes a method `FindActors(`r`)`, which in Dolphin enables applications to pose spatial range queries. Such a query obtains the identifiers and locations of the moving actors that lie inside the spatial range r, where r is a regular quadrilateral $r = \{ (x_{min}, y_{min}), (x_{max}, y_{max}) \} \in \mathbb{R}^2$. We distinguish two phases for query processing:

1. *Partition lookup - shown as Step 1 in Figure 4.1*: The query range r is intersected with spatial partitioning data structures to find which logical cells are spanned by the query.

2. *Search - shown as Step 2 in Figure 4.1*: Indexing actors in the repective logical cells are searched to find the relevant information for actors in r.

Move Processing

The method `Move`(l_d) in the moving actor abstraction updates the current location l_s of a moving actor to a next location l_d. Besides, this movement may asynchronously trigger reactive functions in other moving actors upon satisfaction of the spatial predicates associated to their fences (see below). Similarly to spatial range query processing, we identify two phases for move processing under Actor-based Freshness semantics:

1. *Partition lookup - shown as Step 1 in Figure 4.1*: l_s and l_d are used to intersect with the spatial partitioning data structure to find the partitions corresponding to these locations, namely $cellId_s$ and $cellId_d$, respectively.

2. *Update - shown as Step 2 in Figure 4.1*: The moving actor updates its local state and triggers asynchronous reactions. If $cellId_s$=$cellId_d$, the moving actor calls the indexing actor in this logical cell to record the its update of l_s to l_d. If $cellId_s \neq cellId_d$, the moving actor calls the indexing actor in the logical cell $cellId_d$ to insert l_d, then calls the indexing actor in $cellId_s$ to delete l_s in next round of update.[3] Further steps necessary for reaction processing are described in the following.

Reaction Processing

Each moving actor has a fence f in addition to its location. When a moving actor a calls the method $StartReactiveSensing(p, m)$ in the moving actor abstraction, a will conceptually monitor f with a spatial predicate p – for example, cross, cover, overlap – and process a reaction whenever any other moving actor's itinerary satisfies p with respect to f. The monitoring continues until a invokes the method $StopReactiveSensing()$. A reaction entails executing an application-defined reactive method m in a; m encodes the reactive behavior of a.

[3]The delayed deletion, associated to duplicate elimination in queries, is employed to achieve Freshness semantics (Sidlauskas *et al.*, 2014).

When a is monitoring f with p, Dolphin will update the subscription of a to all monitoring actors in logical cells that overlap its updated fence. This process is performed in two phases:

1. *Partition lookup - shown as Step 1 in Figure 4.1*: The updated fence f of a is intersected with the spatial partitioning data structure to find the logical cells spanned by it.

2. *Subscription update - shown as Step 2 in Figure 4.1*: Monitoring actors in the respective logical cells are contacted to subscribe a to movement in their cells. Subscriptions to monitoring actors that are in logical cells no longer spanned by f are also dropped.

Additionally, when any moving actor a' processes a move, the location update is sent to the relevant monitoring actors for asynchronous processing. This process is performed in two phases:

1. *Partition lookup - shown as Step 1 in Figure 4.1*: $a'.iti$ is intersected with spatial partitioning data structures to find the logical cells spanned by it.

2. *Reaction generation - shown as Step 2 in Figure 4.1*: Monitoring actors in spanned logical cells asynchronously send $a'.iti$ to all moving actors a in their cells whom have called $StartReactiveSensing(p, m)$ (and not yet called $StopReactiveSensing()$) to check if $a.p$ is satisfied over $a.f$.

4.2.2 Constraints and Design Goals for Spatial Partitioning in M-AODBs

To achieve a division of space into logical cells, M-AODBs utilize a spatial partitioning strategy. We call this strategy *logical cell partitioning*. As described in the previous section, logical cell partitioning allows the M-AODB to allocate the processing of distinct operations – and sometimes parts of a single operation – across multiple logical cells. We argue that logical cell partitioning in an M-AODB should satisfy two natural **constraints**:

1. The spatial partitioning strategy must derive a set of logical cells that cover the entire space. This constraint ensures that each moving actor can always be allocated to a logical cell, regardless of its pattern of movement.

2. The logical cells derived by the spatial partitioning strategy cannot be overlapped. This constraint ensures that a moving actor will be allocated to a single logical cell. Since state is encapsulated in actors, we thus avoid replicating actor state, which is complex given that data can only be propagated by asynchronous communication (Bernstein *et al.*, 2017a).

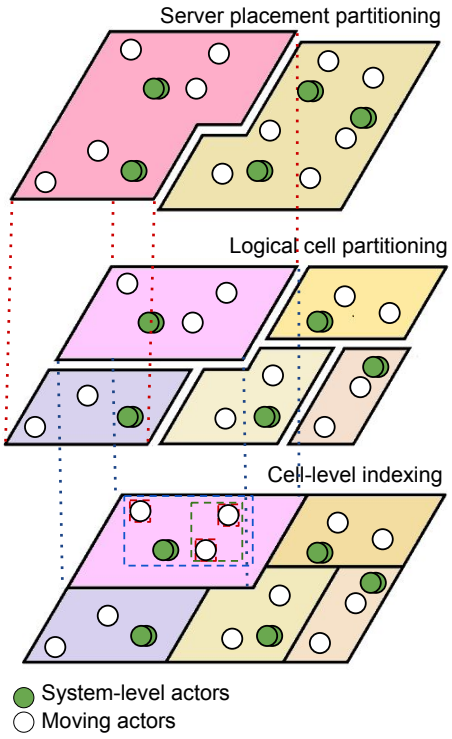


Figure 4.2: Levels of partitioning in Dolphin.

In Dolphin, logical cell partitioning is complemented by two other mechanisms, resulting in a three-level structure as shown in Figure 4.2. The first level is *server placement partitioning*. Here, sets of logical cells are grouped into servers. The second level is logical cell partitioning as discussed above, where logical cells are rectangular shapes. These first two levels allow Dolphin to exploit parallelism across logical cells within a server as well as distribution over multiple servers.

As mentioned previously, moving actors and system-level actors are allocated to logical cells, and a system-level actor of each type – indexing or monitoring actors – supports data management for the moving actors in a given cell. The third level of *cell-level indexing* is implemented by the indexing actors in each cell, and aims at accelerating search operations. Dolphin currently employs in-memory R-trees for this purpose. This chapter focuses on server placement partitioning and logical cell partitioning in M-AODBs, and thus no further discussion of cell-level indexing is presented.

To increase scalability, spatial partitioning in Dolphin is aimed at two main **goals**:

1. For logical cell partitioning, the goal is primarily to balance load across logical cells so as to minimize asynchronous RPC queuing, especially in system-level actors. While communication across logical cells is not irrelevant, its impact is smaller within servers due to shared-memory accesses.

2. For server placement partitioning, the goal is twofold, namely to both balance the load across servers to utilize server capacity and at the same time minimize communication overhead among servers. This is because communication costs in a distributed deployment can become substantial.

To achieve the load balancing aspect of both goals, the ideal is to allocate a similar number of actors in each logical cell or server. On the other hand, we ought to also try to minimize the communication across different logical cells and servers. In the extreme case, if all actors are allocated in a single logical cell, we can eliminate all communication across logical cells, but at the expense of achieving the load balancing goal and deriving a scalable setup.

Note that a good spatial partitioning at the logical cell level should result in most calls between moving actors and system-level actors being inside logical cells, since most interactions in reactive moving object applications are localized in space. When grouping logical cells into servers, it is ideal that this spatial characteristic be preserved. Thus, to reduce communication, we allocate logical cells as a whole, i.e., following *logical cell integrity*, and also respecting spatial locality at the server placement partitioning level. Additionally, we evenly partition logical cells across servers to assist in achieving load balancing in server placement partitioning.

4.3 Spatial Partitioning Techniques in Dolphin

This section first provides a summary of spatial partitioning techniques (Section 4.3.1), then discusses which ones we choose to evaluate based on the constraints and goals for spatial partitioning in M-AODBs (Section 4.3.2), and finally presents how each technique is incorporated in Dolphin (Sections 4.3.3 to 4.3.6).

4.3.1 Summary of Spatial Partitioning Techniques

A great variety of spatial partitioning techniques has been proposed throughout decades of research in spatial database systems (Jones, 1997). In general, methods can be arranged into two broad categories: space-driven partitioning and data-driven partitioning (Rigaux *et al.*, 2002). Space-driven partitioning techniques decompose space into regular or semi-regular shapes that are

indirectly related to data objects. Data objects are then placed into those partitioned shapes based on their spatial features. By contrast, data-driven partitioning techniques identify regions of the space, e.g., coordinates, bounding rectangles, or geometric objects, directly based on data objects. In both categories, a wide range of geometric representations are supported for data objects. However, given our scenario of spatial partitioning in M-AODBs, we assume throughout point data representations.

The space-driven partitioning methods most widely used include the fixed grid (Nievergelt *et al.*, 1984), the Quad-Tree (Finkel and Bentley, 1974), and space-filling curves (SFCs) (Sagan, 2012). In fixed grid partitioning, space is partitioned into regular grids. The choice of the grid size is a critical parameter when discretizing the space into grid cells. Differently, the Quad-Tree recursively partitions space into four equal quadrants until the number of objects in each quadrant does not exceed a predetermined node capacity. As such, the cell size of a Quad-Tree (i.e., the size of leaf nodes) varies depending on the data distribution. Alternatively, in SFCs, space is partitioned by enumerating grid cells that contain at most one object. The visiting order of each such grid cell is the curve value of this cell or object. By doing that, a clustering property is achieved by way of preserving the spatial locality between objects. The two most commonly adopted SFCs are the Hilbert curve (Hilbert, 1935) and the Z-Ordering curve (Morton, 1966).

The K-D-Tree (Bentley, 1975) and the R-Tree (Hadjieleftheriou *et al.*, 2017) are popular data-driven partitioning techniques. K-D-Tree partitioning applies the divide-and-conquer principle to select points to partition the space into regions. Initially, an x coordinate is chosen to partition the whole space into two regions; subsequently, the procedure recursively continues by partitioning sub-regions along another axis until each cell contains no other points. The R-Tree bucketizes data objects into minimum bounding rectangles (MBRs), which are then bucketized into higher-level MBRs recursively. Leaf nodes in the R-Tree group a certain number of data objects, while non-leaf nodes group lower-level nodes in a manner reminiscent of B-Trees.

4.3.2 Discussion Spatial Partitioning Methods Chosen for M-AODB Evaluation

We would like to include in our evaluation of spatial data partitioning in M-AODBs as many as possible of the popular methods as summarized in the previous section, as long as they respect the constraints discussed in Section 4.2.2. As such, we employ four space-driven structures – namely, fixed grid partitioning, Quad-Tree partitioning, Hilbert curve partitioning, and Z-Ordering curve partitioning – and one data-driven structure, K-D-Tree partitioning. The reason why R-Tree partitioning is not included in our evaluation is that R-Tree partitioning in general violates the second of the two constraints presented earlier. Namely, MBRs in the R-tree structure may overlap. We note that there are some R-Tree variants, e.g., the R+Tree (Sellis *et al.*, 1987), R*Tree (Beckmann *et al.*, 1990), and R* Grove (Vu and Eldawy, 2020), that can be adapted to guarantee disjoint partitions at any given level of the R-Tree, but that does not apply across levels. If one attempts to simply use the MBRs of a single level of such an R-Tree variant, then the first of our constraints would be violated, namely that the partitioning must cover the whole space. Properly clipping, expanding, or combining MBRs of such an R-Tree variant to fulfill both constraints of spatial partitioning in M-AODBs is an extension that we leave to future work.

Since the locations of moving actors are dynamic, it is important to consider how to employ the above methods to derive a spatial partitioning to be used by the M-AODB. Importantly, we observe that data distributions in reactive moving object applications are relatively stable. For instance, the vehicle density in a city center is prone to be larger than in a rural area. We assume for the remainder of this work that the variations in distribution that occur over periods of time, e.g., peak time vs. night, do not substantively affect the stability of the spatial distribution. Under this assumption, we conceptually use our spatial partitioning results as a static lookup table for allocating moving actors to partitions or finding the partitions that a moving actor is associated to (see Section 4.2.1). Moreover, in Dolphin, actors are initialized in servers based on server placement partitioning, and re-initialisation of actors is not considered in this chapter. We leave the adoption of periodic updates of such a lookup table to reflect changes over time, e.g,. as discussed by Akdogan et al. (Akdogan *et al.*, 2016), for further work.

4.3.3 Default Partitioning Methods in Dolphin

Fixed grid partitioning is widely adopted in distributed spatial database systems, such as Hadoop-GIS (Aji *et al.*, 2013), SpatialHadoop (Eldawy and Mokbel, 2015; Eldawy and Mokbel, 2013), GeoSpark (Yu *et al.*, 2015; Yu *et al.*, 2016) and SpatialSpark (You *et al.*, 2015). The popularity of fixed grid partitioning stems from the observation that a well-tuned fixed grid is hard to beat in terms of in-memory performance (Sowell *et al.*, 2013; Sidlauskas and Jensen, 2014; Sidlauskas *et al.*, 2009). In the original Dolphin design, this observation from the literature motivated the adoption of this method for logical cell partitioning, since the latter is employed to allocate work within a server.

In short, fixed grid partitioning works as follows. Based on the partition capacity B, i.e., the target number of moving actors per logical cell, the total number of moving actors MA, and the whole space size S, the regular grid size g can be calculated as $g = S/\lceil\sqrt{\lceil MA/B\rceil}\rceil$. To be noticed, we use the partition capacity B to decide the grid size, but the number of moving actors in each grid cell is not necessarily equal to B in fixed grid partitioning. This is due to potential skew in the spatial distribution, which could lead to grid cells with many more actors than the partition capacity (and conversely, cells with many less). This potential partition imbalance is a feature of the fixed grid, since the cell size is not adaptable to different regions of space. After partitioning the whole space into such a regular grid, moving actor locations can be allocated to grid cells based on spatial containment, which achieves logical cell partitioning. An example of Fixed Grid logical cell partitioning is shown in Figure 4.6e.

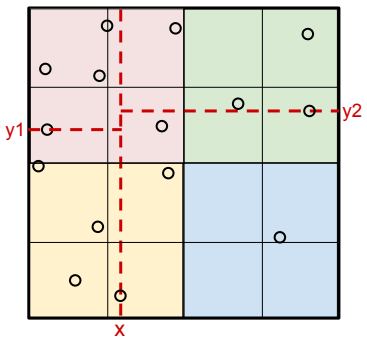


Figure 4.3: Default server placement partitioning grouping.

Dolphin employs by default K-D-Tree partitioning for server placement partitioning. This method respects our spatial partitioning constraints and allows us to decompose the space into any desired number of partitions, which is important for being able to effectively utilize any number of servers. Furthermore, the method has been shown in literature to exhibit competitive performance in a distributed setting, as it adapts to spatial skew (Eldawy *et al.*, 2015; Aji *et al.*, 2015).

Dolphin combines K-D-Tree server placement partitioning with fixed grid logical cell partitioning as follows. Firstly, we partition the whole space by a K-D-Tree structure, but limiting the dividing points based on the number of servers. For instance, in Figure 4.3, to place moving actors into four servers, a dividing point x is used to segregate the whole space into two balanced partitions. Then, in each partition, we choose another dividing point for further partitioning (points $y1$ and $y2$ in the left and right partitions, respectively). After this step, we group logical cells into servers based on these dividing points while respecting the boundaries of logical cells. By way of example, in Figure 4.3, if the position of the left boundary of a logical cell is equal to or smaller than x, then this logical cell will be grouped into the left partition; and if the upper boundary of a logical cell is greater than y, the logical cell will be grouped into upper partition. Following this procedure, logical cells will be assigned to different servers, as shown in different colors in Figure 4.3. This procedure of K-D-Tree partitioning respects the natural spatial proximity of logical cells. Additionally, K-D-Tree partitioning adapts to skew in a way that is ignored by fixed grid partitioning, thus addressing costly inter-server communication costs. An example of default server placement partitioning, which corresponds to the logical cell partitioning in Figure 4.6e, is shown in Figure 4.6j.

4.3.4 SFC Partitioning in Dolphin

As known from previous research, the Hilbert curve has a higher spatial clustering performance than Z-Ordering (Abel and Mark, 1990; Rong and Faloutsos, 1991; Jagadish, 1990), making the Hilbert curve the primary choice of space-filling curve for spatial partitioning. For the sake of comparison, however, we implement both Hilbert and Z-Ordering in Dolphin for logical cell partitioning. In both cases of SFC partitioning, the whole space is conceptually partitioned into regular grids that contain at most one object. We define the curve value of each moving actor cv as the visiting order of each grid. Then, moving actors are packed into logical cells $\lfloor cv/B \rfloor$ based on curve value and partition capacity B. Hilbert and Z-Ordering logical cell partitioning examples are shown in Figures 4.6a and 4.6b, respectively.

After applying the partitioning techniques to the logical cell partitioning level, we achieve a partitioning of the whole space into M logical cells such that $M = \lceil MA/B \rceil$. We then group logical cells based on their number as given

by logical cell generation (see above) so that $= \lceil M/N \rceil$ consecutive adjacent logical cells are grouped into a server, where N is the number of servers. After grouping all logical cells, server placement partitioning is achieved. The latter partitioning also preserves spatial proximity among server-level partitions, as the logical cell numbers can be seen as a coarsening of the original moving actor curve values. An example of Hilbert and Z-Ordering server placement partitioning, which corresponds to the logical cell partitioning in Figure 4.6a and Figure 4.6b, are shown in Figures 4.6f and 4.6g, respectively.

4.3.5 K-D-Tree Partitioning in Dolphin

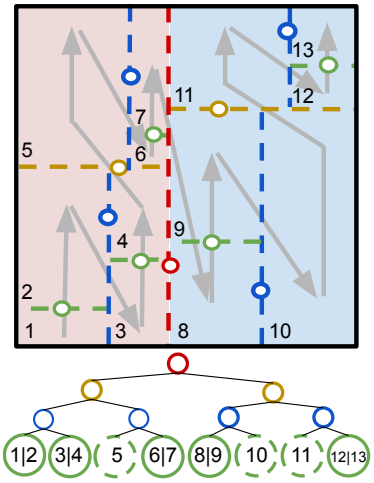


Figure 4.4: K-D-Tree partitioning example.

A K-D-Tree splits the whole space into leaf-node regions containing no data. We derive logical cells from nodes in a level from a balanced K-D-Tree. Here, the level is chosen to be $\lceil \log_2 \lceil MA/B \rceil \rceil$ so as to create partitions with size close to the partition capacity B. If this level is the maximum level of K-D-Tree, the regions induced by leaf nodes become the logical cells, as show in Figure 4.4; otherwise, for each node in this level, the regions induced by its left-child node with corresponding children nodes are grouped as a logical cell, and the same is done for the right-child node. Since the splits created by a K-D-Tree always intersect a point from the input, the partitioning derived for the space is based on the data. We number logical cells based on the Z-Ordering curve. By doing that, cells that are adjacent in space end up keeping proximity in the numbering. An example of K-D-Tree logical cell partitioning is shown in Figure 4.6c.

After logical cell partitioning is performed based on the K-D-Tree, we obtain a number M of logical cells. Since spatial proximity is preserved by logical cell numbering, we can apply the grouping strategy of SFC to calculate a server placement partitioning. Namely, we get a server placement partitioning by grouping consecutive logical cells so that each of N servers contains $\lceil M/N \rceil$ logical cells. For instance, as shown in Figure 4.4, if there are two servers, logical cells numbered 1 to 7 are grouped into server 1 (colored in red), while logical cells 8 to 13 are grouped into server 2 (colored in blue). An example of

K-D-Tree server placement partitioning, which corresponds to the logical cell partitioning in Figure 4.6c, is shown in Figure 4.6h.

4.3.6 Quad-Tree Partitioning in Dolphin

Quad-Tree partitioning as implemented in Dolphin splits the whole space into leaves with maximum capacity B. A logical cell is created for each such leaf. These logical cells are numbered based on the Z-Ordering curve by way of bit interleaving as illustrated in Figure 4.5. The latter makes the numbering sensitive to the partitioning depth found by the Quad-Tree. Note that the numbering preserves proximity of adjacent logical cells. an example of Quad-Tree logical cell partitioning is shown in Figure 4.6d.

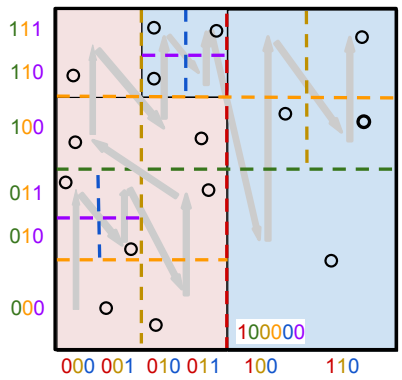


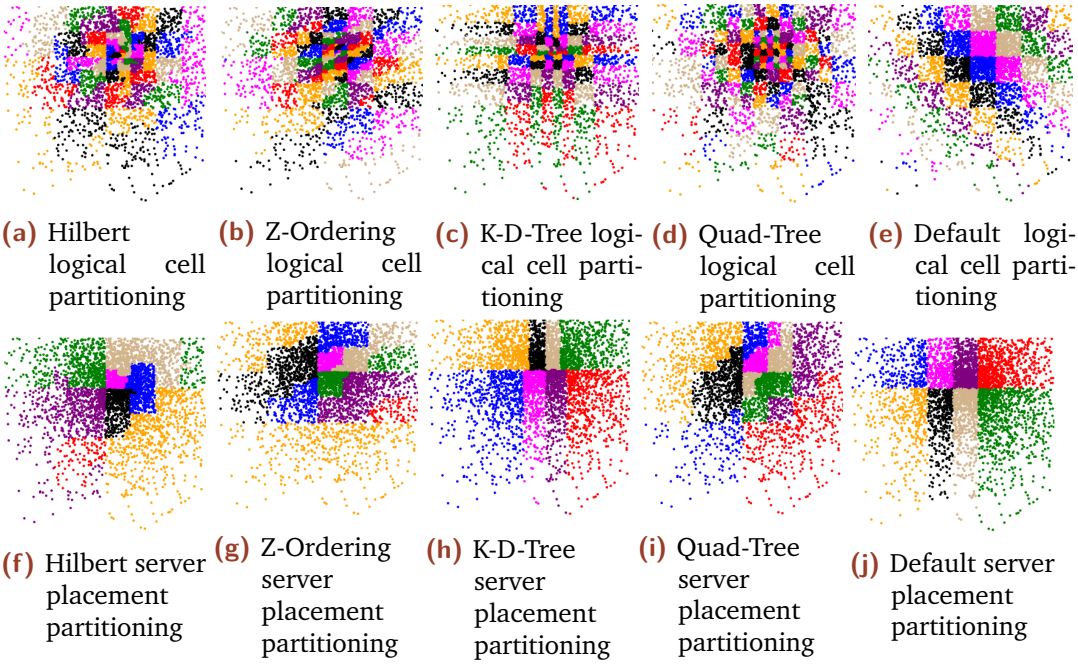
Figure 4.5: Quad-Tree partitioning example.

After logical cell partitioning by the Quad-Tree, we get a number M of logical cells. Once again, we derive a server placement partitioning by following the clustering strategy of SFC, where $\lceil M/N \rceil$ logical cells are placed into each of N servers. For instance, as shown in Figure 4.5, if there are two servers, the first to tenth logical cells are grouped into server 1 (colored in red), and the eleventh to nineteenth logical cells are grouped into server 2 (colored in blue). An example of Quad-Tree server placement partitioning, which corresponds to the logical cell partitioning in Figure 4.6d, is shown in Figure 4.6i.

4.4 Experimental Evaluation

This section presents our experimental evaluation of the implemented spatial partitioning techniques in light of several significant features of M-AODBs. In particular, the goals of the evaluation are as follows:

1. Investigate the effect of partition capacity and determine proper settings of partitioning capacity for different spatial partitioning techniques (Section 4.4.2).



(a) Hilbert logical cell partitioning (b) Z-Ordering logical cell partitioning (c) K-D-Tree logical cell partitioning (d) Quad-Tree logical cell partitioning (e) Default logical cell partitioning

(f) Hilbert server placement partitioning (g) Z-Ordering server placement partitioning (h) K-D-Tree server placement partitioning (i) Quad-Tree server placement partitioning (j) Default server placement partitioning

Figure 4.6: Examples of partitioning methods in Dolphin for logical cell partitioning and server placement partitioning (Number of moving actors=5000, Space size= $10km^2$, Number of Hotspots=1, Partition capacity=100, Number of servers=8).

2. Explore the trade-offs in the choice of spatial partitioning technique in Dolphin for varied reactive sensing ratio, query intensity, and reaction range scenarios (Sections 4.4.2, Section 4.4.2, and Section 4.4.2, respectively).

3. Evaluate how spatial partitioning techniques affect scalability under skewed data (Section 4.4.2).

4. Illustrate how partitioning techniques affect move and reaction performance under varied degrees of spatial skew. (Section 4.4.2).

4.4.1 Experimental Setup

As mentioned previously, our experimental evaluation builds on Dolphin, a first implementation of an M-AODB leveraging Microsoft Orleans. In the original proposal of Dolphin in Chapter 3, a comprehensive experimental setup was employed, covering cloud-based deployment, adaptation of benchmarks and workloads from the moving objects literature, and datasets with varied spatial skew. We adopt broadly the same experimental setup as this previous work; however, a few adaptations are pursued to focus on skewed data and explore additional parameters. We document these adaptations in the following, as well as briefly review the general setup to make this chapter as self-contained

as possible. We refer the reader to previous work for additional details and justifications for the various settings in Chapter 3.

Cloud Service and Deployment

In our experiments, Orleans silos (Microsoft, 2021) are used as servers to host all actors and run all application logic. We deploy all silos in separate instances on AWS EC2 (*Amazon EC2 - Secure and resizable compute capacity to support virtually any workload* 2020) and store the membership table for the silos in Amazon DynamoDB (*Amazon DynamoDB - Fast and flexible NoSQL database service for any scale* 2021). One c5.xlarge instance is used for synchronizing benchmark client threads, and up to another eight of the same type of instance are used as silos. Client threads are run in a c5.4xlarge instance. All instances run on Windows Server 2019 Base and Orleans 3.1.2. They are placed in one cluster deployment group (Amazon, 2021) and the same subnet (Zou *et al.*, 2011) to reduce the physical network latency.

Benchmarks, Workloads, and Datasets

We implement a Gaussian distribution benchmark to investigate the performance of Dolphin over datasets with varied settings of spatial skew. In the Gaussian distribution benchmark, hotspots are uniformly distributed in space, and moving actors are then distributed around these hotspots under a Gaussian distribution. A moving actor moves faster when it is closer to the hotspot (Sowell *et al.*, 2013).

Our experiments start by initializing a set of moving actors. Each client thread continuously sends either move or spatial range query requests, depending on the experiment, without interruption. As discussed before, reactions are triggered in the system asynchronously by move requests. Client threads are set so that enough load is generated to fully saturate the servers. We have observed that servers run stably at high CPU utilization in all experiments. As in previous work in Chapter 3, we primarily experiment with moves and reactions, unless otherwise stated, to focus on reactive behaviors.

As discussed in Section 4.2.1, all of the categories of spatial data processing supported by Dolphin, namely spatial range queries, moves, and reactions, make use of spatial data partitioning structures. We implement the latter as an immutable lookup table given our assumption of stability in spatial data distribution. The immutability of the structure allows us to easily replicate it

Table 4.1: Benchmark, Workload, and Dataset Parameters.

Parameters	Gaussian Distribution Benchmark
Number of Servers	1, 2, 4, 8
Client Threads per Server	8 (single server) or 4 (multiple servers)
Number of Moving Actors	5000 per server
Space Size (km^2)	100, 200, 400, 800
Partition Capacity	10, 25, 50, 100, 250, 500 moving actors per logical cell
Data Skewness	5000/500/100/50/5/1 moving actors per hotspot
Fence Size (m^2)	500×500, 707×707, **1000×1000**, 1414×1414, 2000×2000
Query Size (m^2)	1000×1000
Max Speed (km/h)	80
Reactive Sensing Percentage (%)	0,**12.5**,25,50,100
Query Ratio (%)	**0**,20,40,60,80,100

in each silo. We leave the investigation of techniques to update our spatial partitioning lookup table for future work.

The datasets we used for the experiments correspond to the workloads described above. As mentioned previously, we keep the data aligned with the experiments in the original Dolphin proposal in Chapter 3.

The parameters used for benchmarks, workloads, and datasets are shown in Table 4.1, where default settings are highlighted in bold when applicable.

Single- vs. Multi-Server Experiments

We chose to run the experiments in Sections 4.4.2, 4.4.2, 4.4.2, and 3.5.2 on a single server, since the effect of the parameters we test can already be observed well in the absence of distributed data placement. To stress the behavior of the system under an extreme case of spatial skew, we chose the most skewed Gaussian data distribution, i.e., the one with only one hotspot in space, in single-server experiments. Furthermore, in these experiments all actors are activated in one server, thus obviating server placement partitioning. Since only logical cell partitioning is used, for clarification, we name the partitioning methods employed `Fixed Grid`, `Hilbert`, `Z-Ordering`, `K-D-Tree`, and `Quad-Tree`. Recall that these methods are discussed in Sections 4.3.3 to 4.3.6.

Experiments that test scalability and vary spatial skew – namely, the ones in Sections 4.4.2 and 4.4.2 – are conducted in multiple servers. In these sections, we refer to `Default` as the partitioning method of Section 4.3.3 instead of `Fixed Grid`, since server placement partitioning is also employed.

Metrics and Measurement Methodology

To characterize the performance of Dolphin utilizing different spatial partitioning methods, we primarily measure client request throughput, composed of moves and spatial range queries, and reaction throughput. These metrics are obtained by an epoch-based measurement approach. In more detail, client threads submit requests to the system when an epoch starts and stop doing so when the epoch ends. After the end of the epoch, client threads can record how many move and query tasks were sent, how many tasks have been finished, latencies of finished tasks, along with other data. Each epoch is set to be 10 seconds. A measurement for an experiment is run for ten epochs, including three warm-up epochs.

Because reactions are asynchronously triggered by moves in Dolphin rather than sent as client requests, we cannot measure reaction throughput directly from the client side. We thus record reactions and their latencies in every moving actor. After synchronizing all client threads, the client calls all moving actors to collect the reaction numbers and latencies in the whole system in this epoch. When this process is completed, another epoch can start.

Task throughput is a calculated as an average, along with its standard deviation, of per-epoch measurements aggregated across all client threads, except for warm-up epochs. *Reaction throughput* is calculated in a similar fashion as task throughput. *Move latency* and *query latency* are measured as the time from the moment a moving actor receives a move or query request to the time this move or query has finished execution in this moving actor. *Task latency* is the end-to-end latency of either move or query requests measured at the client side. *Client-server latency* represents the communication and queuing overhead between client and server, which is the task latency exclusive of move or query latency. While we focus on reporting both throughput and latency as averages with standard deviation as error bars, we have also collected the 25%, 50%, 90%, 99% percentile latencies as well as maximum task and reaction latencies for validation purposes.

In addition to throughput and associated latency distribution measurements, we delve down into observed performance issues by calculating further metrics related to load balance and communication. We define the *communication from moving actors to system-level actors* as the total number of messages that are sent from moving actors to both indexing and monitoring actors divided by the number of non-warm-up epochs. Since indexing actors never

perform asynchronous RPCs to moving actors, the complement to the former metric is the *communication from monitoring actors to moving actors*, which is defined as the total number of disseminated messages from monitoring actors to their subscribed reactive moving actors divided by the number of non-warm-up epochs. Similarly, we further define an efficiency metric to capture how many disseminated messages from monitoring actors are required to generate one reaction on average, which we term the *monitoring-to-moving-actor communication per reaction*. The *move load balance* is the coefficient of variation of the communication from moving actors to system-level actors under a workload consisting of 100% moves and no queries, and it aims to capture how well the load directly generated by moves is spread between logical cells. Note that a number close to zero in this metric indicates near-perfect load balance, while higher numbers indicate imbalanced load.

4.4.2 Experimental Results

What is the impact of partition capacity on different spatial partitioning techniques?

The partition capacity B, i.e., the target number of moving actors per logical cell, influences the amount of communication between moving actors and system-level actors as well as the intensity of the load placed on system-level actors. For smaller B, the load on system-level actors in each logical cell is smaller, in general, but the resulting finer-grained partitioning can introduce more communication across logical cells. By contrast, coarse-grained partitioning increases the burden on system-level actors, but potentially lessens communication intensity. For various spatial partitioning methods in Dolphin, we would like to observe the relationship between partition capacity and system performance so as to derive guidelines for the choice of spatial partitioning methods under a range of different partition capacities.

We tested all spatial partitioning methods implemented in Dolphin and varied the partition capacity B from 10 to 500 under Actor-Based Freshness semantics. As can be seen in Figures 4.7 and 4.8, to achieve the best move and reaction throughputs, different spatial partitioning methods would need to be configured with distinct partition capacity. If we compare the performance of these methods under their best setting for the partition capacity B, we can observe that the `Fixed Grid` exhibits the best move and reaction throughput overall. However, the variance in both metrics is much larger for the `Fixed Grid` than

for other methods when we enlarge the partition capacity (cf. B =10, B =250, and B =500). In other words, spatial partitioning methods other than the `Fixed Grid`, especially the `Quad-Tree` and the `K-D-Tree`, are more resilient to the setting chosen for partition capacity.

To explain the variation of throughput across methods for different settings of B, we profiled Dolphin and remarked that a large contributor to observed performance is the cost of communication. We analyze two primary types of communication among actors in Dolphin: communication from moving actors to system-level actors and communication from monitoring actors to moving actors.The former communication component is shown in Figure 4.9a. Here, we see that the the differences in this communication component between B =10 and B =500 are 2.123x, 2.133x, 1.861x, 1.914x, and 1.272x for `Hilbert`, `Z-Ordering`, `K-D-Tree`, `Quad-Tree`, and `Fixed Grid`, respectively. The reason for additional communication with smaller partition capacity is that fine-grained partitions lead to more interactions across logical cells. However, this effect is not reflected in move throughput, in which the differences are 1.065x, 1.068x, 1.043x, 1.147x, and 1.166x between B =10 and B =500 for the same methods listed above.

To explain this reverse trend in move throughput compared with communication between moving actors and system-level actors, we analyze the other major component of communication, namely communication from monitoring actors to moving actors. This latter metric is shown in Figure 4.9b. We observe a general trend of higher dissemination from monitoring actors to reactive moving actors as the partition capacity is increased. Also, we notice that the monitoring dissemination number dominates the total amount of communication. The reason for these effects is that in the current system implementation, only spatial partitioning helps restrict which reactive monitoring actors receive monitoring information. In other words, monitoring information regarding the movement of actors is distributed to all reactive moving actors in relevant partitions. Thus, the amount of monitoring information disseminated is in proportion to the number of reactive moving actors in each such relevant partition. Additionally, under the same reactive sensing rate, more and more moving actors are contained in a given partition as we enlarge the partition size. This increases the number of moves that need to be disseminated and eventually filtered by reactive moving actors.

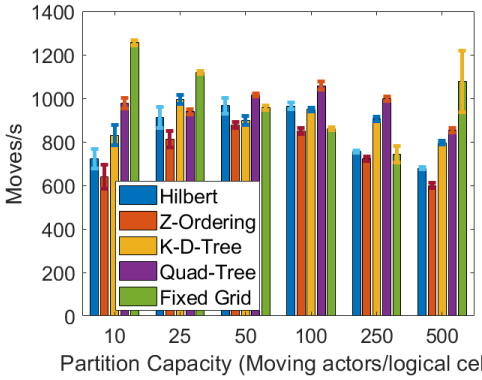


Figure 4.7: Move throughputs under varied partition capacities with different partitioning methods

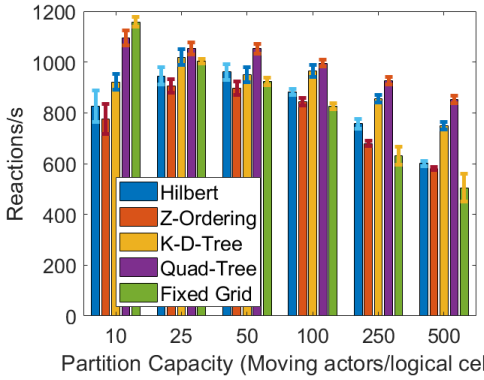
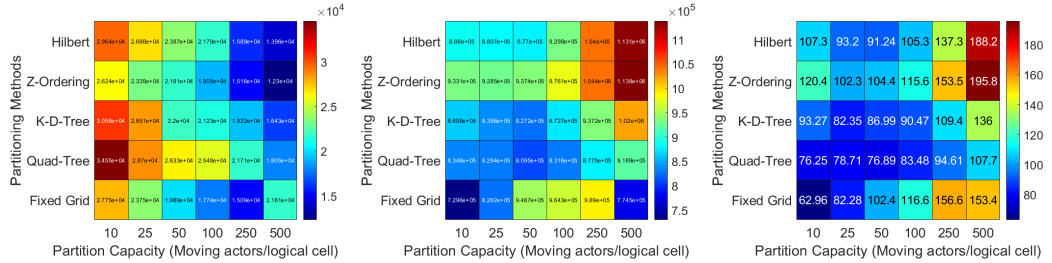


Figure 4.8: Reaction throughputs under varied partition capacities with different partitioning methods

To fully characterize the latter observation, we report in Figure 4.9c the monitoring-to-moving-actor communication per reaction. We can observe a strong correlation between this metric and the observed throughput, especially of moves. This result indicates that there is significant asynchronous processing of monitoring information sent to reactive moving actors with no chance to trigger reactions, and that this processing takes valuable system resources that could otherwise have been utilized to increase throughput.

As a final remark, in this experiment, we can see parameter tuning is necessary for fixed grid partitioning. After fine-tuning the partition capacity B, the fixed grid is the best choice, which agrees with its use in many popular spatial databases (Yu *et al.*, 2015; Yu *et al.*, 2016; You *et al.*, 2015; Aji *et al.*, 2013; Eldawy and Mokbel, 2015; Eldawy and Mokbel, 2013). However, other spatial partitioning methods are more resilient to the tuning of the partition capacity parameter. In particular, when partition capacity may be hard to tune, then `Quad-Tree` and `K-D-Tree` are recommended.

For our following experiments, we employ the best observed setting for partition capacity B for different partitioning methods, i.e., B=50 for the two SFC methods, B=25 for `K-D-Tree`, B=100 for `Quad-Tree`, and B=10 for fixed grid partitioning.



(a) Communication from moving actors to system-level actors under varied partition capacities

(b) Communication from monitoring actors to moving actors under varied partition capacities

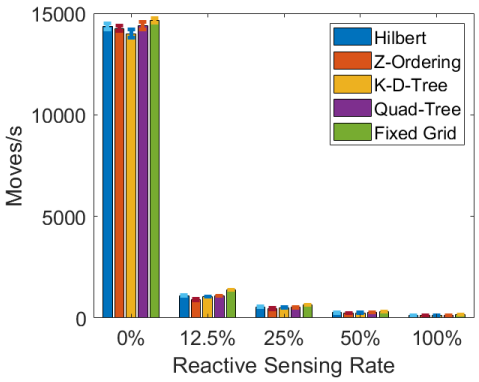
(c) Monitoring-to-moving-actor communication per reaction under varied partition capacities

Figure 4.9: The impact of partition capacity B in Dolphin with different partitioning methods.

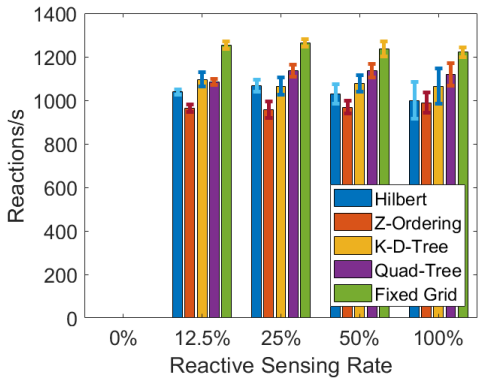
How does reactive sensing rate affect system performance under different partitioning methods?

In the previous section, we have noted that dissemination of monitoring information in logical cells represents a substantial cost and drives the throughput of Dolphin. The amount of this type of communication that is generated by the system is related to both the numbers of moving actors and reactive moving actors in each logical cell. Since we fix the partition capacity B, the number of reactive moving actors per logical cell becomes the main variable, which is itself primarily driven by the reactive sensing rate. Thus, we would like to investigate how the system behaves with increasing reactive sensing intensity under the best partition capacity observed for each partitioning technique.

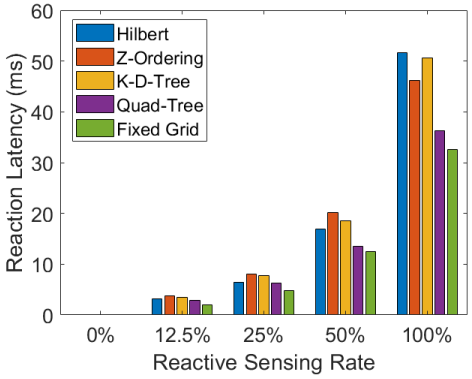
Move and reaction throughput generated by different spatial partitioning techniques under varied reactive sensing rate is shown in Figures 4.10a and 4.10b, respectively. The move throughputs of all partitioning methods decrease dramatically with increasing reactive sensing rate in the system, while the reaction throughputs keep stable. This trend in move throughput is consistent with a corresponding substantial increase in reaction latencies, which we can see in Figure 4.10c, since as observed previously there is interference between reaction generation and move throughput. Move task latencies, shown in Figure 4.10d, further confirm this observation. Here, we can see a large increase in client-server latencies, indicating that the system is experiencing queuing from concurrent load. Lower move throughput will imply that reactions are triggered relatively less frequently, but the growing reactive sensing rate increases the number of reactive moving actors. The balance between these two competing factors results in the reaction throughput remaining stable.



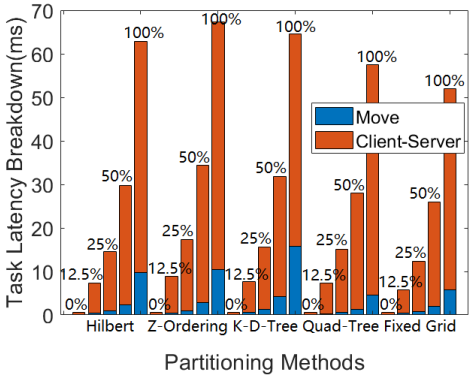
(a) Move throughput under varied reactive sensing rate with different partitioning methods



(b) Reaction throughput under varied reactive sensing rate with different partitioning methods



(c) Reaction latencies under varied reactive sensing rate with different partitioning methods



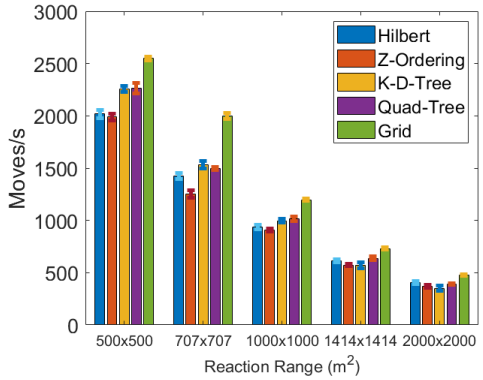
(d) Task latency breakdown under varied reactive sensing rate with different partitioning methods

Figure 4.10: The impact of reactive sensing rate on system performance with different partitioning methods.

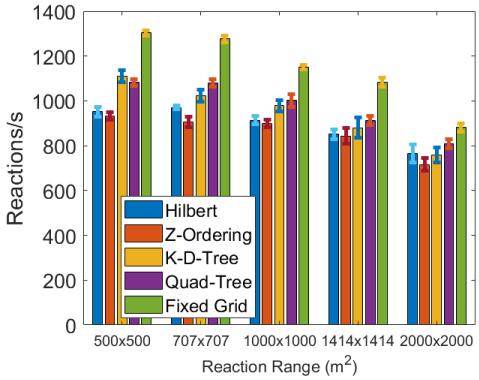
We can also see that among the spatial partitioning techniques, the most resilient to increasing reactive sensing rates were the `Quad-Tree` and the `Fixed Grid`, both of which exhibit relatively lower reaction latencies than other methods. At 100% reactive sensing rate, these methods exhibit reaction latencies of 36.319 ms and 32.558 ms, respectively, while the latencies of other methods exceed 46.193 ms. Therefore, `Quad-Tree` and `Fixed Grid` are recommended for higher reactive sensing rates due to their better observed reaction latency.

How does reaction range affect system performance under different partitioning methods?

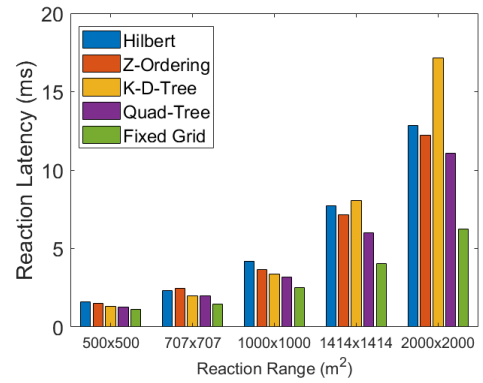
Based on our observations in the experiments varying partition capacity and reactive sensing rate, we conjecture that other variables that affect the amount



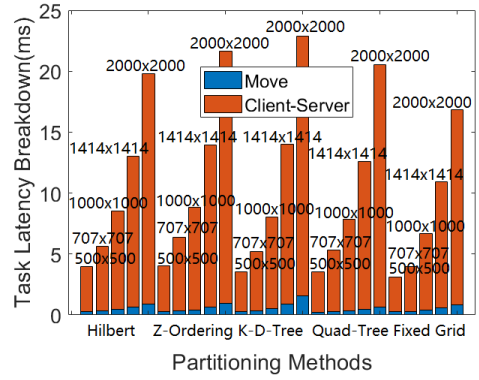
(a) Move throughput under varied reaction ranges with different partitioning methods



(b) Reaction throughput under varied reaction ranges with different partitioning methods



(c) Reaction latencies under varied reaction ranges with different partitioning methods



(d) Task latencies breakdown under varied reaction ranges with different partitioning methods

Figure 4.11: The impact of reaction range on system performance with different partitioning methods.

of information disseminated from monitoring actors to moving actors may impact move or reaction throughput. A natural question is whether the reaction range would have such an effect, since the larger the range, the more monitoring actors in nearby logical cells a given reactive moving actor needs to subscribe to. Consequently, it becomes easier for reactions to be triggered because of the larger coverage of space containing potentially more moving actors. We conduct experiments on a set of reaction ranges – 500m×500m, 707m×707m, 1000m×1000m, 1414m×1414m, and 2000m×2000m – that are meaningful to be used in real scenarios.

As expected, Figures 4.11a shows that move throughput decreases with larger reaction ranges for all spatial partitioning methods. Perhaps surprisingly, reaction throughput also declines slightly with expanding reaction ranges, as can be seen in Figure 4.11b. This slight decrease stands in contrast to the

stable reaction throughput observed when we varied the reactive sensing rate, suggesting that the expected increase in reactions triggered by a larger range is partly offset by the expected reduction in reactions triggered due to lower move throughput.

Figures 4.11c and 4.11d also confirm that the system behaves in a similar way as observed above when we varied reactive sensing rate, indicating that overload leads reaction and move task latencies to soar. Similarly to what was observed in the original Dolphin proposal (cf. Chapter 3), we expect that this overload situation could be addressed by by employing a server with larger capacity. For brevity, we leave a validation experiment to future work.

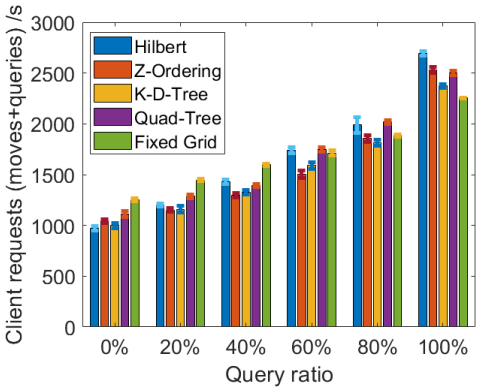
We further remark that `K-D-Tree` is not recommended for large reaction range settings, because the reaction latency increased 13.013x when the size of the reaction range increased by 16x. Meanwhile, only a 5.45x increase was seen for `Fixed Grid`, from 1.141ms to 6.223ms. Thus, for large reaction range scenarios, `Fixed Grid` is recommended because it provides tighter reaction latency, while `K-D-Tree` suffers the most.

How does query ratio affect system performance under different partitioning methods?

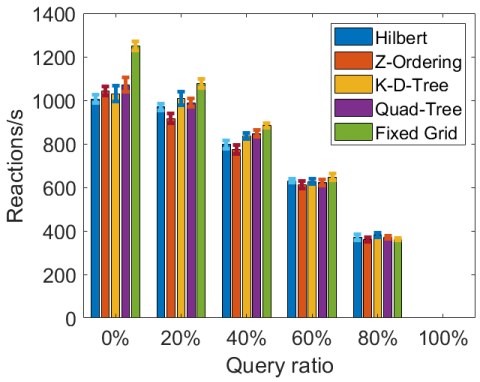
Workloads in M-AODBs may not only include move requests from clients, but also spatial query requests. To observe how Dolphin behaves under spatial skew with various combinations of spatial range query and move requests, we vary the ratio between queries and moves to which the system is exposed. We explore a wide range of query ratios, going from 0% queries and 100% moves to 100% queries and 0% moves.

In Figure 4.12a, we can see that the task throughput, which includes both queries and moves, increases for all spatial partitioning methods as the query ratio is increased. This trend is opposite to one observed in the original Dolphin proposal under a uniform data distribution in Section 3.5.2, wherein the task throughput decreased with increase query ratio. We observed then that reaction latencies were kept stable, and the dominant effect was that spatial range query requests are more costly than move requests.

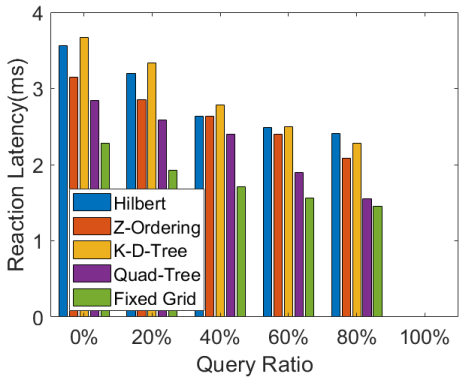
With a skewed distribution as in the present experiment, we see a different dynamic is revealed by reaction and task latencies, shown in Figures 4.12c and 4.12d, respectively. Reaction latencies drop, indicating that load in the



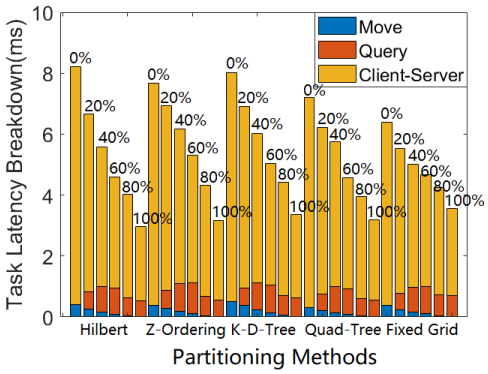
(a) Move throughputs under varied query ratios with different partitioning methods



(b) Reaction throughputs under varied query ratios with different partitioning methods



(c) Reaction latencies under varied query ratios with different partitioning methods



(d) Task latency breakdown under varied query ratios with different partitioning methods

Figure 4.12: The impact of query ratio on system performance with different partitioning methods.

system is getting reduced. The effect is confirmed by the drop in the client-server component of task latencies. At the same time, it is still verified that the spatial range queries take more time to process at the server side than move requests for all partitioning methods (cf. move and query components of task latencies in Figure 4.12d). However, the decrease in load is sufficiently large to release resources that offset the extra cost of queries and boost task throughput.

Additionally, from Figures 4.12a and 4.12b, we remark that the move and reaction throughputs of `Fixed Grid`, the method exhibiting the best performance under a pure move workload, are taken over by the other four spatial partitioning methods as query ratio is increased. Furthermore, the `Quad-Tree` catches up on reaction latencies with the `Fixed Grid` when the query ratio becomes bigger.

Can system leverage different partitioning methods to achieve scalability under spatial skew?

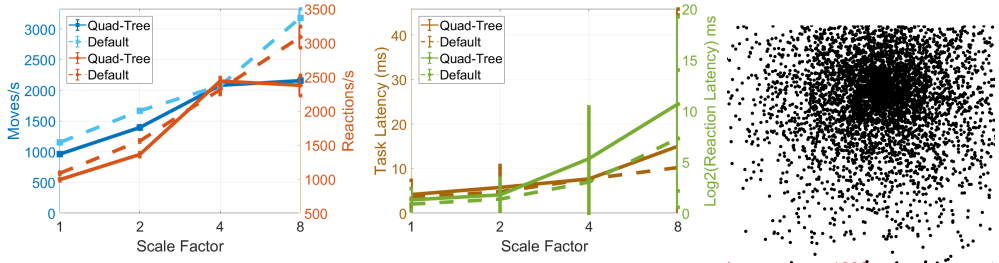
The spatial partitioning methods evaluated above in Dolphin exhibit similar behaviors across a range of parameters in a single-server setting. For multi-server experiments, we focus our attention on two representative methods. As the first method, we employ the default partitioning implementation of Dolphin. It includes `Fixed Grid` for logical cell partitioning, which performed well on most experiments after partition capacity tuning, and extends it with K-D-Tree server placement partitioning while respecting logical cell boundaries (see Section 4.3.3). Second, we select `Quad-Tree`, which among the remaining methods performed best overall and was also more resilient to the specific tuning of partition capacity. It is extended multiple servers by Z-Ordering grouping (see Section 4.3.6). For simplicity, we refer to the two methods as `Default` and `Quad-Tree`, respectively, in the remainder.

We first vary scale factors from 1 to 8, where each increase in scale factor is associated with proportionally expanded numbers of server numbers, client requests, dataset sizes, and numbers of hotspots. We can see from Figure 4.13a that Dolphin does not scale linearly with the scale factor under data distributions with spatial skew. In particular, when the scale factor increases 8x, move throughput only rises by 2.761x and 2.246x for `Default` and `Quad-Tree`, respectively. Correspondingly, reaction throughput respectively increases by

2.842x and 2.394x. Both move task and reaction latencies, shown in Figure 4.13b, also increase with scale factor, and particularly so when the scale factor doubles from 4 to 8. This effect is more pronounced for the `Quad-Tree` than for `Default`.

To characterize the above effects, we first argue that the load imposed in the system with higher scale factors is increased superlinearly, and thus linear scalability cannot be expected in this scenario. In more depth, Figure 4.13c illustrates the data distribution for scale factor equal to 1, where there is one hotspot and 5000 moving actors that are arranged around it following a Gaussian distribution. Suppose that we kept on adding hotspots with the same number of actors each that are fully disjoint in space as we also increased the number of servers. In this hypothetical case, we would expect load on Dolphin to increase linearly by 8x when the scale factor would be 8. By contrast, the actual spatial distribution we encounter at scale factor 8 is shown in Figure 4.13d. The space size and the number of actors are proportionally enlarged by the scale factor, and the same amount of moving actors are placed around each hotspot following the same distribution. However, since hotspots are placed randomly in space, they do not end up being disjoint, but actually overlapping. This implies that there are regions of higher density that can create dramatic increases in load even if the reaction range remains unchanged.

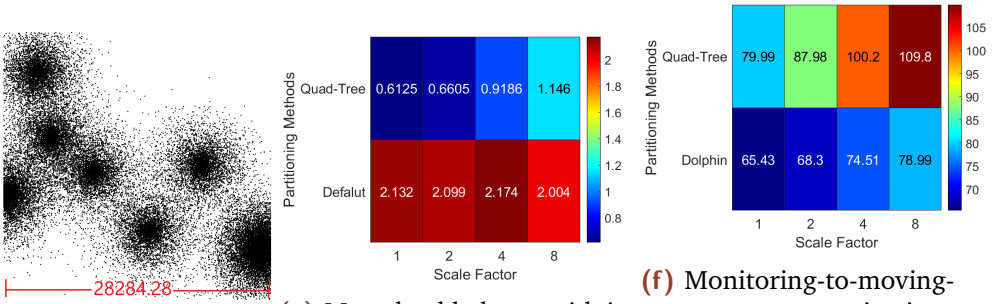
To further delve into the differences between `Default` and `Quad-Tree`, we display the move load balance coefficient with scale factor in Figure 4.13e and the monitoring-to-moving-actor communication per reaction metric in Figure 4.13f. The move load balance coefficient of `Default` is relatively stable (e.g., increases of 0.9443x, 1.040x, and 0.922x between scale factors 1-2, 2-4, and 4-8, respectively). By contrast, move load balance for `Quad-Tree` increases by 1.871x from scale factor 1 to 8. Additionally, the efficiency in monitoring information dissemination is worse in the `Quad-Tree` than it is on `Default`. The combination of the two factors indicates that the `Quad-Tree` will suffer more than `Default` as the scale factor is increased, even if neither method could scale linearly.



(a) Move and reaction throughputs with increasing servers and workload under Quad-Tree and Default partitioning

(b) Move and reaction latencies with increasing servers and workload under Quad-Tree and Default partitioning

(c) Visualization of data distribution in one server



(d) Visualization of data distribution in eight servers

(e) Move load balance with increasing servers and workload under Quad-Tree and Default partitioning

(f) Monitoring-to-moving-actor communication per reaction with increasing servers and workload under Quad-Tree and Default Partitioning

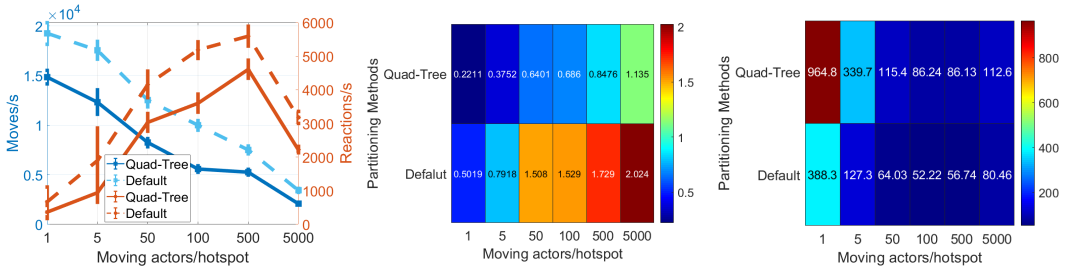
Figure 4.13: System scalability when employing Quad-Tree and Default partitioning.

How does the system behave under varied spatial skew with different partitioning methods?

We also test how different spatial partitioning techniques can help with performance under various spatially skewed distributions. We run experiments from near-uniform data (1 moving actor per hotspot) to heavily skewed data (5000 moving actors per hotspot).

For both `Quad-Tree` and `Default`, as seen in Figure 4.14a, move throughput declines along with the increase in spatial skew. As pointed out in the previous section, more skewed data can lead to worse move load balance, which is confirmed in Figure 4.14b for both methods. However, we remark that `Default` is more resilient to the changes in skew. The move throughput of `Default` decreases by 5.625x, while the move throughput of `Quad-Tree` declines by 7.163x across the whole range of skew settings. In Figure 4.14b, we further observe that `Default` has worse move load balance coefficients across all skew settings than those of the `Quad-Tree`. However, the move load balance coefficient degrades across the skew settings by 4.040x for `Default`, but by as much as 5.133x for the `Quad-Tree`. The latter suggests that the `Quad-Tree` is more affected by increasing skew. Notwithstanding, to fully explain the move throughput differences between the methods, the overhead of communication due to reaction processing needs to be taken into account. Figure 4.14c contrasts the two methods with the monitoring-to-moving-actor communication per reaction metric. The results suggest that the interference from reaction processing on system resources available for ingesting moves is significantly higher for the `Quad-Tree` than for `Default`, reinforcing the effect seen for move load balance.

Reaction throughput goes up with increasing skew up to the 500 moving actors/hotspot distribution, then falls off when data distribution gets denser. We reason that with a denser data distribution, one move would be more likely to generate more reactions, because more moving actors may have been affected by this move. However, the drop in move throughput counterbalances this trend. In particular, if the frequency of movement is reduced, so must be the frequency of triggering reactions. Still, compared with the throughputs for both moves and reactions in the original Dolphin proposal in Section 3.5.2, this result underscores the need of careful tuning to data distribution for `Default`, which achieves higher numbers, as well as the possibility to utilize



(a) Move and reaction throughputs under varied spatial skew with Quad-Tree and Default partitioning.

(b) Move load balance under increasing spatial skew distribution with Qua-Tree and Default partitioning

(c) Monitoring-to-moving-actor communication per reaction under varied spatial skew with Quad-Tree and Default Partitioning

Figure 4.14: System performance under varied spatial skew settings with Quad-Tree and Default partitioning.

methods that require less tuning to deliver competitive performance, such as the `Quad-Tree`.

4.5 Related Work

In most spatial databases, spatial data partitioning generates a set of partitions that are processing units for spatial processes, which not only aids in load balancing, but also helps with parallelization of computations. As observed by Zhou et.al (Zhou *et al.*, 1998), it is critical to preserve spatial locality when designing a data partitioning framework for spatial parallel processing. Distributed spatial databases that are currently widely used employ a variety of such spatial partitioning techniques to achieve parallelism. For instance, Hadoop-GIS (Aji *et al.*, 2013) uses recursive grid partitioning and a global directory (R-tree, R*-tree) that helps identify partitions that need to be loaded. Aji et al. (Aji *et al.*, 2015) evaluate a wide range of spatial partitioning algorithms, including fixed grid, Hilbert curve, Binary split, Strip, Sort-tile-recursive, and Boundary optimized strip, and observe their effects on partition balance, query performance, and partition efficiency in the context of this system. Also, this work points out that finding a optimal spatial partitioning is NP-hard. It is thus argued that an efficient partitioning choice is based on practical requirements and application scenarios. Similarly, SpatialHadoop (Eldawy and Mokbel, 2015; Eldawy and Mokbel, 2013) supports grid files, R-tree, and R+-trees to index partitions across all nodes. Eldawy et al. (Eldawy *et al.*, 2015) experimentally evaluate other four alternative partitioning techniques, namely Quad-Tree, K-D-Tree, Z-Ordering, and Hilbert, together with the three default

partitioning techniques in SpatialHadoop. The results reveal the impact of these different partitioning methods on spatial query performance as well as the partitioning structure quality obtained in SpatialHadoop. GeoSpark (Yu *et al.*, 2015; Yu *et al.*, 2016) allows fixed grid, Hilbert, R-tree, and Voronoi partitioning. SpatialSpark (You *et al.*, 2015) employs fixed grid partitioning, binary space partitioning, as well as tile partitioning. LocationSpark (Tang *et al.*, 2016) employs grids and Quad-Trees to ensure data partition are balanced in data size. GeoMesa (Hughes *et al.*, 2015) utilizes geo-hashing based on Z-Ordering to derive a spatio-temporal partitioning. MD-HBase (Nishimura *et al.*, 2013) builds K-D-Tree and Quad-Tree partitioning on top of HBase by utilizing Z-curve linearization. Even though various spatial partitioning techniques are used in a number of spatial data management solutions to aid in parallel processing and data distribution, none of those systems has investigated the impact of partitioning techniques on reactivity and reactive systems, as our work does.

Dolphin is the first implementation of recently proposed Moving Actor-Oriented Databases (cf. Section 3.4). Dolphin relies by default on a fixed grid for logical cell partitioning, which is complemented by server placement partitioning based on a K-D-Tree. Previous experiments in Dolphin have not included a detailed investigation of a variety of spatial partitioning techniques to evaluate their potential to aid in handling data skewed in space, which is discussed in this chapter.

There exist specialized partitioning methods that we can explore in the future. For instance, Anwar et al. (Anwar *et al.*, 2014) and Ji and Geroliminis (Ji and Geroliminis, 2012) propose specific traffic congestion-based spatial partitioning methods for urban transportation networks. Deep learning has been investigated as a possible methodology to automatically choose an optimal partitioning technique based on a specific spatial data distribution (Vu *et al.*, 2020). In a similar vein, instead of relying only on classic partitioning techniques, a potential avenue for future work is the exploration of learned data structures as they hold potential to adapt to a particular dataset and workload (Kraska *et al.*, 2018). ZM-index (Wang *et al.*, 2019a), ML-index (Davitkova *et al.*, 2020), LISA (Li *et al.*, 2020a), PolyFit (Li *et al.*, 2020b), Flood (Nathan *et al.*, 2020), TRS-Tree (Wu *et al.*, 2019), and XIndex (Tang *et al.*, 2020) are all methodologies that extend learned data structures to spatial data. Some of these methodologies are limited, e.g., offering only constrained support for spatial queries and datatypes or not offering support for dynamic workloads.

Additionally, and more importantly, none of them have been implemented and tested in a reactive spatial system.

4.6 Conclusions and Future Work

This chapter has evaluated classic spatial partitioning techniques in Dolphin, the first implementation of recently proposed Moving Actor-Oriented Databases (M-AODBs) for reactive moving object applications. In particular, the evaluation has focused on the behavior of these methods in Dolphin when faced with data distributions with spatial skew, which is typical in reactive moving object applications. Spatial skew can cause load imbalances in parallel processing as well as increase communication overheads. To explore how spatial partitioning techniques can help handle problems caused by spatially skewed data in Dolphin, we firstly analyzed the multiple levels of spatial partitioning used in Dolphin and discussed the associated goals and constraints that need to be respected by spatial partitioning methods. Secondly, we integrated several classic partitioning methodologies into Dolphin. Thirdly, we conducted experiments in Dolphin on the basis of important parameters to delineate the impact of different partitioning methods on varied performance metrics. Our results identify how to choose adequate partitioning techniques based on use scenarios. Additionally, we also reveal that increasing the efficiency of dissemination of monitoring information in Dolphin is likely to substantially impact system performance, thus being a worthy direction for future study.

5 Conclusion

Due to the unprecedented popularization of Internet-of-Things (IoT) applications, new challenges in the construction of IoT data platforms have emerged: 1) an IoT data platform needs to be scalable because of the explosive growth of IoT data and expanding user requests; 2) the heterogeneity of IoT data formats and arbitrary degree of distribution of IoT devices requires an IoT data platform to be flexible; 3) the dynamicity in deployment of IoT devices raises the need for elasticity in an IoT data platform; 4) a new requirement of tight latency reactivity has been put forward, especially with the advent of mobile IoT applications; and, 5) providing programmability in an IoT data platform is crucial to ease the burden of application developers in designing complex application functionality with associated data management features.

There are a number of solutions that can help with building IoT data platforms. Spatial data management is an essential part of managing mobile IoT data, but many spatial data management systems fail to provide reactivity functionality, which is by contrast supported in reactive systems. However, reactive systems have limited and constrained support for spatial features. Meanwhile, Actor-Oriented Databases (AODBs) are a recently proposed cloud-based data management solution for distributed, interactive, and scalable applications, and these new systems are a natural to fit the requirements of building IoT data platforms. However, AODBs lack support for reactivity and spatial data management.

Based on our analysis of the requirements, characteristics, and constraints of IoT data platforms, we provided guidance on how to model IoT data platforms with AODBs. Furthermore, we presented a new proposal detailing how to enrich an AODB with spatial data management features and reactivity functionality to support building scalable and reactive data platforms for mobile IoT applications.

5.1 Summary of the Dissertation

There are three main contributions in this dissertation:

1. **Modeling and Building IoT Data Platforms with Actor-Oriented Databases.** In the first part of this dissertation, we discussed the challenges of building IoT data platforms, including how to support and manage a massive amount of concurrently generated heterogeneous data from IoT applications, how to effectively share and protect those data, and how to achieve system scalability. We investigated two distinct real-world IoT application case studies, namely beef cattle tracking and tracing and structural health monitoring (SHM). These case studies helped us illustrate functional and non-functional requirements of IoT data platforms. We further remarked that Actor-Oriented Databases (AODBs) are naturally suitable for addressing the challenges and fulfilling the requirements of IoT data platforms. However, how to utilize AODBs to model and build IoT data platforms to manage data from IoT applications is an open research question. Therefore, we investigated and provided guidelines on how to model IoT data platforms with AODBs, including: how to identify actors from IoT entities; how to decide the granularity of actors; how to enforce relationship constraints across actors; and how to navigate the trade-offs between employing actors or non-actor objects for frequently accessed IoT entities. A prototype of an SHM data platform based on our suggested guidance was built and validated through experiments.

2. **Building a Moving Actor-Oriented Database for Reactive Mobile IoT Applications.** To provide an effective and programmable data platform for reactive mobile IoT applications, in this part of the dissertation, we introduced a novel system that combines scalable spatial data management with reactivity. As part of this proposal, we defined a new abstraction of moving actors that provides fundamental functionalities needed by moving objects in reactive mobile IoT data platforms. We term our novel system architecture supporting moving actors Moving Actor-Oriented Databases (M-AODBs), underscoring that it builds upon the substrate given by AODBs. To tackle a core challenge in realizing M-AODBs, we defined two concurrency semantics, namely Actor-based Freshness semantics and Actor-based Snapshot semantics, satisfying different use scenarios. Furthermore, we provided the first implementation of M-AODBs called Dolphin based on the Microsoft Orleans virtual actor runtime. Experiments on synthetic distribution and C-ITS benchmarks showed that Dolphin can satisfy the mobile IoT data platform requirements of providing for spatial data management, scalability, and low-latency reactions.

3. **Evaluating Partitioning Techniques to Handle Spatial Skew in Dolphin.** Dolphin achieves scalability and provides low-latency reactions based on parallelism, distribution, and appropriate concurrency semantics. However, spatial skew is typical in mobile IoT applications, which can affect the load balance and communication overhead between partitions. In our Dolphin experiments, we noticed that spatial skew heavily affected system performance. To understand and quantify the impact of spatial partitioning in Dolphin, we conducted a thorough evaluation of several classic partitioning techniques in Dolphin in this part of the dissertation. First, we analyzed the partitioning architecture of Dolphin, illustrating the goals and constraints at different levels of Dolphin partitioning. Then, we discussed the integration of classic spatial partitioning methods into Dolphin and conducted an evaluation of essential factors that affect system performance. Based on an analysis of our results, we discussed how to choose the most adequate partitioning techniques under various scenarios. Furthermore, our experiments revealed that the dominant factor that influences Dolphin performance is the overhead of disseminating monitoring information inside logical cells.

This dissertation aimed to provide scalable and reactive data management for mobile IoT applications based on Actor-Oriented Databases. To achieve this goal, we firstly clarified the problems, challenges, and requirements of IoT data platforms and provided clear guidance on modeling IoT data platforms based on AODBs. Secondly, we proposed the novel architecture of Moving Actor-Oriented Databases that provides scalability and supports low-latency reactions for reactive moving object applications. Dolphin, the first prototype of an M-AODB, is a concrete implementation that can help alleviate the complexity of development and data management for this challenging scenario. Lastly, we conducted a thorough evaluation of how classic partitioning methods can help with handling spatial skew in Dolphin.

5.2 Ongoing and Future Work

Given the enormous potential impact of mobile IoT applications on society, we hope that our work can serve the role of a catalyst to encourage the research community to explore further novel data management solutions, programming abstractions, and algorithms for this scenario. We present our current ongoing work and list several promising directions for future work in this section:

1. **Guarantee Fault Tolerance and Failure Recovery in M-AODBs.** The Orleans runtime dynamically manages actor life-cycle and provides fault tolerance on actor instances, but recovery of the system to a consistent state in the presence of failures is not given by Orleans. To achieve failure recovery in M-AODBs, it is necessary to take into consideration how to persist and reload actor state during processing and reactivations.

2. **Integrate Historical Data Processing and Management to M-AODBs.** Even though M-AODBs employ data storage as a back-end, historical data processing and management were not in our consideration. In the future, support for queries on historical data or reactions based on historical data analysis comprise a promising direction for enriching M-AODBs to meet even more of the requirements of mobile IoT applications.

3. **Enforce Data Constraints between Actors in AODBs.** Due to encapsulated state in actors and actors only influencing each other by asynchronous messaging, relationship constraints between actors are actually distributed even though in some situations data changes need to be consistent. It remains as an interesting direction for further exploration to develop special-purpose transaction or workflow models to guarantee these constraints across actors are respected.

4. **Reduce Stream Dissemination Costs by Indexing during Reaction Processing.** Our evaluation shows that the communication from monitoring actors to their subscribed moving actors in each logical cell is the main bottleneck exhibited by Dolphin. Additionally, we have observed that the number of messages sent is much larger than the number of reactions that are eventually triggered. Implementing indexing on the monitoring range of reactive moving actors, and sending monitoring information based on indexing is a feasible way to solve the issue. However, reaction monitoring areas of moving actors are dynamic, changing along with their movements. Indexing the dynamic reaction monitoring areas of moving actors while respecting concurrency semantics in an asynchronous distributed system is a nontrivial direction of further exploration.

5. **Employ Dynamic Partitioning Techniques in M-AODBs.** In this work, we assume the spatial distribution of moving actor locations is relatively stable even though data is dynamic. For instance, a hotspot in space is always considered to be denser than other areas most of the time. For some applications, this assumption may not hold. These applications may require

dynamic partitioning that accommodates highly dynamic data where the skew in the spatial distribution of moving actors changes over long periods of time. Therefore, dynamic partitioning techniques can be evaluated, and potentially adopted, to improve performance in these scenarios.

6. **Utilize Learned Partitioning Techniques in M-AODBs.** Recent research has explored the idea of employing a learned data structure to derive a spatial index that fits a given dataset. Instead of employing a general-purpose classic partitioning structure, recent studies on learned data structures suggest that we can capture spatial properties of specific data distributions, and tailor a partitioning to adapt to this data distribution. However, learned partitioning techniques have not been implemented or evaluated in a reactive IoT data platform, which is a worthy path for future research in M-AODBs.

Bibliography

6

Abd El-Aziz, Eman Mohamed, Saleh Mesbah, and Khaled Mahar (2012). „GIS-based decision support system for criminal tracking“. In: *Proceedings of the 2012 22nd International Conference on Computer Theory and Applications (ICCTA)*. IEEE, pp. 30–34.

Abel, David J and David M Mark (1990). „A comparative analysis of some two-dimensional orderings“. In: *International Journal of Geographical Information Systems* 4.1, pp. 21–31.

Adya, Atul, Robert Gruber, Barbara Liskov, and Umesh Maheshwari (1995). „Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks“. In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. ACM Press, pp. 23–34.

Agha, Gul A. (1990). *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press.

Agricultural Robotics: The Future of Robotic Agriculture (2020). `https://arxiv.org/ftp/arxiv/papers/1806/1806.06762.pdf`.

Aji, Ablimit, Hoang Vo, and Fusheng Wang (2015). „Effective Spatial Data Partitioning for Scalable Query Processing“. In: *CoRR* abs/1509.00910. arXiv: `1509.00910`.

Aji, Ablimit, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz (2013). „Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce“. In: *Proceedings of the 39th international conference on Very large data bases (VLDB)*. 6.11, pp. 1009–1020.

Akbar, Adnan, Abdullah Khan, Francois Carrez, and Klaus Moessner (2017). „Predictive analytics for complex IoT data streams“. In: *IEEE Internet of Things Journal* 4.5, pp. 1571–1582.

Akdogan, Afsin, Cyrus Shahabi, and Ugur Demiryurek (2016). „D-ToSS: A Distributed Throwaway Spatial Index Structure for Dynamic Location Data“. In: *IEEE Trans. Knowl. Data Eng.* 28.9, pp. 2334–2348.

Akka - Build Powerful Reactive, Concurrent, and Distributed Applications More Easily (2020). `https://akka.io/docs/`.

Akka Documentation, Version 2.5.17, IoT example use case (2018). https://doc.akka.io/docs/akka/2.5/guide/tutorial.html.

Akka Streams version 2.5.18 (2018). https://doc.akka.io/docs/akka/2.5/stream/.

Albani, Dario, Joris IJsselmuiden, Ramon Haken, and Vito Trianni (2017). „Monitoring and mapping with robot swarms for agricultural applications". In: *Proceedings of 14th IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS*. IEEE Computer Society, pp. 1–6.

Ali, Mohamed H., Badrish Chandramouli, Balan Sethu Raman, and Ed Katibah (2010a). „Real-time spatio-temporal analytics using Microsoft StreamInsight". In: *Proceedings of the 18th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, (ACM-GIS)*. ACM, pp. 542–543.

Ali, Mohamed H., Badrish Chandramouli, Balan Sethu Raman, and Ed Katibah (2010b). „Spatio-Temporal Stream Processing in Microsoft StreamInsight". In: *IEEE Data Eng. Bull.* 33.2, pp. 69–74.

Amazon EC2 - Secure and resizable compute capacity to support virtually any workload (2020). https://aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc.

Anand, TM, K Banupriya, M Deebika, A Anusiya, T Anand, K Banupriya, and A Anusiya (2015). „Intelligent transportation systems using iot service for vehicular data cloud". In: *International Journal for Innovative Research in Science and Technology* 2.2, pp. 80–86.

Andzic, Mladen, Mike Ray, Mike Ray, Steve Stein, and David Coulter (2020). *Microsoft Spatial Library*. https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-data-types-overview?view=sql-server-ver15.

Annex, Andrew (2018). *GeoWave Provides Geospatial and temporal Indexing on top of Accumulo and HBase*. https://github.com/locationtech/geowave.

Anwar, Tarique, Chengfei Liu, Hai Le Vu, and Christopher Leckie (2014). „Spatial Partitioning of Large Urban Road Networks". In: *Proceedings of the 17th International Conference on Extending Database Technology, (EDBT)*. OpenProceedings.org, pp. 343–354.

Apache Kafka (2021). https://kafka.apache.org/.

Apache Samza - A distributed stream processing framework (2021). http://samza.apache.org/.

Apache Spark Streaming (2021). https://spark.apache.org/streaming/.

Apache Storm (2021). https://storm.apache.org/.

Azure Queue (AQ) Stream Provider (Jan. 2019). https://dotnet.github.io/orleans/Documentation/streaming/stream_providers.html?q%3Dqueue%23azure-queue-aq-stream-provider%0A.

Arasteh, Hamidreza, Vahid Hosseinnezhad, Vincenzo Loia, Aurelio Tommasetti, Orlando Troisi, Miadreza Shafie-khah, and Pierluigi Siano (2016). „Iot-based smart cities: A survey“. In: *Proceedings of the IEEE 16th International Conference on Environment and Electrical Engineering, EEEIC 2016*. IEEE, pp. 1–6.

Atten, Christophe, Loubna Channouf, Grégoire Danoy, and Pascal Bouvry (2016). „UAV Fleet Mobility Model with Multiple Pheromones for Tracking Moving Observation Targets“. In: *Proceedings of the Applications of Evolutionary Computation - 19th European Conference, EvoApplications 2016*. Vol. 9597. Lecture Notes in Computer Science. Springer, pp. 332–347.

Atzori, Luigi, Antonio Iera, Giacomo Morabito, and Michele Nitti (2012). „The social internet of things (siot)–when social networks meet the internet of things: Concept, architecture and network characterization“. In: *Computer networks* 56.16, pp. 3594–3608.

Aulbach, Stefan, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger (2008). „Multi-tenant databases for software as a service: schema-mapping techniques“. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008*. ACM, pp. 1195–1206.

Amazon Web Service (2020). https://aws.amazon.com/.

Amazon DynamoDB - Fast and flexible NoSQL database service for any scale (2021). https://aws.amazon.com/dynamodb/.

AWS IoT Core (2018). https://aws.amazon.com/iot-core/.

Amazon (2021). *AWS Placement groups*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

Amazon Relational Database Service (Nov. 2018). https://aws.amazon.com/rds/.

Bandyopadhyay, Debasis and Jaydip Sen (2011). „Internet of things: Applications and challenges in technology and standardization“. In: *Wireless Personal Communications* 58.1, pp. 49–69.

Beckmann, Norbert, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger (1990). „The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles“. In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. ACM Press, pp. 322–331.

Bentley, Jon Louis (1975). „Multidimensional binary search trees used for associative searching“. In: *Communications of the ACM* 18.9, pp. 509–517.

Bernstein, Philip A. (2018). „Actor-Oriented Database Systems“. In: *Proceedings of the 34th IEEE International Conference on Data Engineering, ICDE 2018*. IEEE Computer Society, pp. 13–14.

Bernstein, Philip A., Sebastian Burckhardt, Sergey Bykov, *et al.* (2017a). „Geo-distribution of actor-based services“. In: *Proceedings of the ACM Program. Lang.* 1.OOPSLA, 107:1–107:26.

Bernstein, Philip A. and Sergey Bykov (2016). „Developing Cloud Services Using the Orleans Virtual Actor Model“. In: *IEEE Internet Comput.* 20.5, pp. 71–75.

Bernstein, Philip A., Mohammad Dashti, Tim Kiefer, and David Maier (2017b). „Indexing in an Actor-Oriented Database“. In: *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017*.

Bernstein, Philip A, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin (2014). „Orleans: Distributed virtual actors for programmability and scalability“. In: *MSR-TR-2014–41*.

Bhowmick, Subhajit, Palash Kumar Kundu, and Dharma Das Mandal (2021). „IoT Assisted Real Time PPG Monitoring System for Health Care Application“. In: *Proceedings of the 2021 IEEE Second International Conference on Control, Measurement and Instrumentation, CMI 2021*. IEEE, pp. 122–127.

Billinghurst, Mark, Adrian J. Clark, and Gun A. Lee (2015). „A Survey of Augmented Reality“. In: *Found. Trends Hum. Comput. Interact.* 8.2-3, pp. 73–272.

Bocher, Erwan, Gwendall Petit, Nicolas Fortin, and Sylvain Palominos (2015). „H2GIS a spatial database to feed urban climate issues“. In: *ICUC9*.

Bonér, Jonas, Dave Farley, Roland Kuhn, and Martin Thompson (2014). „The reactive manifesto“. In: *Dosegljivo: http://www. reactivemanifesto. org/.[Dostopano: 21. 08. 2017]*.

Bonfort, Thomas and Thomas Bonfort (2013). „MapCache: The Fast Tiling Server From The MapServer Project“. In:

Borggren, Kasper Myrtue (2018). „Scalable Structural Health Monitoring Data Platform using Actors as a Database“. MA thesis. Copenhagen Denmark: University of Copenhagen.

Bowers, Shawn and Bertram Ludäscher (2005). „Actor-oriented design of scientific workflows“. In: *Proceedings of the International Conference on Conceptual Modeling, ER 2005*. Springer, pp. 369–384.

Breen, Thomas B (Feb. 2009). *System and method for updating geo-fencing information on mobile devices*. US Patent 7,493,211.

Brinkhoff, Thomas (2002). „A Framework for Generating Network-Based Moving Objects“. In: *GeoInformatica* 6.2, pp. 153–180.

Bureau, The U.S. Census (2020). *U.S. Census Bureau - TIGER/Line*. `https://www.census.gov/cgi-bin/geo/shapefiles/index.php`.

Burgard, Wolfram, Mark Moors, Dieter Fox, Reid Simmons, and Sebastian Thrun (2000). „Collaborative multi-robot exploration“. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*. Vol. 1. IEEE, pp. 476–481.

Bussche, Arnaud Vanden (2020). *Hedging technolog risks in public C-ITS investments: a real options approach*. `http://technoeconomics.idlab.ugent.be/education/ThesisArnaudVandenBussche.pdf`.

Bykov, Sergey, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin (2011). „Orleans: cloud computing for everyone“. In: *Proceedings of the ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC 2011*. ACM, p. 16.

Calbimonte, Jean-Paul, Óscar Corcho, and Alasdair J. G. Gray (2010). „Enabling Ontology-Based Access to Streaming Data Sources“. In: *Proceedings of the The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010*. Vol. 6496. Lecture Notes in Computer Science. Springer, pp. 96–111.

Cao, Hung and Monica Wachowicz (2020). „A Holistic Overview of Anticipatory Learning for the Internet of Moving Things: Research Challenges and Opportunities“. In: *ISPRS Int. J. Geo Inf.* 9.4, p. 272.

Capponi, Andrea, Claudio Fiandrino, Burak Kantarci, Luca Foschini, Dzmitry Kliazovich, and Pascal Bouvry (2019). „A Survey on Mobile Crowdsensing Systems: Challenges, Solutions, and Opportunities“. In: *IEEE Commun. Surv. Tutorials* 21.3, pp. 2419–2465.

Carbone, Paris, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas (2015). „Apache flink: Stream and batch processing in a single engine“. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4.

Chandrasekaran, Sirish and Michael J. Franklin (2004). „Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams“. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004*. Morgan Kaufmann, pp. 348–359.

Chansanchai, Athima (2018). *Pegasus II mission sends balloon high above Earth and invites you along for an Internet of Things ride*. https://news.microsoft.com/features/pegasus-ii-mission-sends-balloon-high-above-earth-and-invites-you-along-for-an-internet-of-things-ride/.

Chen, Liang, Sarang Thombre, Kimmo Järvinen, Elena Simona Lohan, Anette Alén-Savikko, Helena Leppäkoski, M Zahidul H Bhuiyan, Shakila Bu-Pasha, Giorgia Nunzia Ferrara, Salomon Honkala, *et al.* (2017). „Robustness, security and privacy in location-based services for future IoT: A survey". In: *IEEE Access* 5, pp. 8956–8977.

Chen, Su, Christian S. Jensen, and Dan Lin (2008). „A benchmark for evaluating moving object indexes". In: *Proceedings of the 34TH international conference on Very large data bases, VLDB 2008* 1.2, pp. 1574–1585.

Cimino, Chiara, Elisa Negri, and Luca Fumagalli (2019). „Review of digital twin applications in manufacturing". In: *Computers in Industry* 113, p. 103130.

Cimino, Mario G. C. A., Beatrice Lazzerini, Francesco Marcelloni, and Andrea Tomasi (2005). „Cerere: an information system supporting traceability in the food supply chain". In: *Proceedings of the 7th IEEE International Conference on E-Commerce Technology Workshops, CEC 2005 Workshops*. IEEE Computer Society, pp. 90–98.

Clementini, Eliseo and Paolino Di Felice (1995). „A comparison of methods for representing topological relationships". In: *Information sciences-applications* 3.3, pp. 149–178.

Costa, Rui (2018). „The Internet of Moving Things [Industry View]". In: *IEEE Technol. Soc. Mag.* 37.1, pp. 13–14.

Cui, Jin, Giedre Sabaliauskaite, Lin Shen Liew, Fengjun Zhou, and Biao Zhang (2019). „Collaborative Analysis Framework of Safety and Security for Autonomous Vehicles". In: *IEEE Access* 7, pp. 148672–148683.

Davitkova, Angjela, Evica Milchevski, and Sebastian Michel (2020). „The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries". In: *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020*. OpenProceedings.org, pp. 407–410.

Deoliveira, Justin (2008). „GeoServer: uniting the GeoWeb and spatial data infrastructures". In: *Proceedings of the 12th Global Spatial Data Infrastructure Association World Conference, GSDI 2020*, pp. 25–29.

Derler, Patricia, Edward A. Lee, and Alberto L. Sangiovanni-Vincentelli (2012). „Modeling Cyber-Physical Systems". In: *Proc. IEEE* 100.1, pp. 13–28.

Dindar, Nihal, Peter M. Fischer, Merve Soner, and Nesime Tatbul (2011). „Efficiently correlating complex events over live and archived data streams“. In: *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011*. ACM, pp. 243–254.

Du, Jiaqing, Sameh Elnikety, and Willy Zwaenepoel (2013). „Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks“. In: *Proceedings of the IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013*. IEEE Computer Society, pp. 173–184.

Duckett, Tom, Simon Pearson, Simon Blackmore, and Bruce Grieve (2018). „Agricultural Robotics: The Future of Robotic Agriculture“. In: *CoRR* abs/1806.06762. arXiv: 1806.06762.

Eldawy, Ahmed, Louai Alarabi, and Mohamed F. Mokbel (2015). „Spatial Partitioning Techniques in Spatial Hadoop“. In: *Proceedings of the 41st international conference on Very large data bases, VLDB 2015* 8.12, pp. 1602–1605.

Eldawy, Ahmed and Mohamed F. Mokbel (2013). „A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data“. In: *Proceedings of the 39th international conference on Very large data bases, VLDB 2013* 6.12, pp. 1230–1233.

Eldawy, Ahmed and Mohamed F. Mokbel (2015). „SpatialHadoop: A MapReduce framework for spatial data“. In: *Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE 2015*. IEEE Computer Society, pp. 1352–1363.

Eldeeb, Tamer and Phil Bernstein (Oct. 2016). *Transactions for Distributed Actors in the Cloud*. Tech. rep. MSR-TR-2016-1001. Microsoft Research.

Elrefaei, Lamiaa A, Alaa Alharthi, Huda Alamoudi, Shatha Almutairi, and Fatima Al-rammah (2017). „Real-time face detection and tracking on mobile phones for criminal detection“. In: *Proceedings of the 2nd International Conference on Anti-Cyber Crimes, ICACC 2017*. IEEE, pp. 75–80.

Brazilian Agricultural Research Corporation - A Embrapa (2018). https://www.embrapa.br/en/international.

Erlang-Build massively scalable soft real-time systems (2020). https://www.erlang.org/docs.

ETSI EN 302 890-2 V2.1.1 (2020-10) (2020). https://standards.iteh.ai/catalog/standards/etsi/4d2e31b5-3bd5-406c-92c3-ae3a977e7cbf/etsi-en-302-890-2-v2.1.1-2020-10.

Event Hubs - Simple, secure, and scalable real-time data ingestion (2021). https://azure.microsoft.com/en-us/services/event-hubs/.

Finkel, Raphael A. and Jon Louis Bentley (1974). „Quad trees a data structure for retrieval on composite keys“. In: *Acta informatica* 4.1, pp. 1–9.

Fizza, Kaneez, Abhik Banerjee, Karan Mitra, Prem Prakash Jayaraman, Rajiv Ranjan, Pankesh Patel, and Dimitrios Georgakopoulos (2021). „QoE in IoT: a vision, survey and future directions“. In: *Discover Internet of Things* 1.1, pp. 1–14.

Flores-Martin, Daniel, Javier Rojo, Enrique Moguel, Javier Berrocal, and Juan M Murillo (2021). „Smart Nursing Homes: Self-Management Architecture Based on IoT and Machine Learning for Rural Areas“. In: *Wireless Communications and Mobile Computing* 2021.

Furieri, Alessandro (2014). „SpatiaLite“. In: *linha]. Disponível [Acedido: 30-Nov-2015]*.

Future Cropping partnership website (2018). https://futurecropping.dk/.

Gaede, Volker and Oliver Günther (1998). „Multidimensional Access Methods“. In: *ACM Comput. Surv.* 30.2, pp. 170–231.

Galic, Zdravko, Emir Meskovic, and Dario Osmanovic (2017). „Distributed processing of big mobility data as spatio-temporal data streams“. In: *GeoInformatica* 21.2, pp. 263–291.

Google IoT Core (Jan. 2018). `https://cloud.google.com/iot-core/`.

Grain Placement - Mircosoft Orleans (2020). `https://dotnet.github.io/orleans/Documentation/grains/grain_placement.html`.

Global Standards One (2018). `https://www.gs1.org/`.

Gubbi, Jayavardhana, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami (2013). „Internet of Things (IoT): A vision, architectural elements, and future directions“. In: *Future generation computer systems* 29.7, pp. 1645–1660.

Gupta, Munish (2012). *Akka Essentials*. Packt Publishing Ltd.

Gutierrez, Lucio, Eleni Stroulia, and Ioanis Nikolaidis (2012). „fAARS: a platform for location-aware trans-reality games“. In: *International Conference on Entertainment Computing*. Springer, pp. 185–192.

Güting, Ralf Hartmut, Victor Teixeira de Almeida, and Zhiming Ding (2006). „Modeling and querying moving objects in networks“. In: *Proceedings of the 32nd international conference on Very large data bases, VLDB 2006* 15.2, pp. 165–190.

Guting, Ralph and Markus Schneider (2005). *Moving Objects Databases*. Morgan Kaufmann.

H2 Database Engine (2021). `https://www.h2database.com/html/main.html`.

Hadjieleftheriou, Marios, Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J. Tsotras (2017). „R-Trees: A Dynamic Index Structure for Spatial Searching“. In: *Encyclopedia of GIS*. Springer, pp. 1805–1817.

Haller, Philipp (2012). „On the integration of the actor model in mainstream technologies: the scala perspective“. In: *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012*. ACM, pp. 1–6.

Halo: Combat Evolved (2021). `https://www.halowaypoint.com/en-us`.

Hanson, Eric N., Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park, and Albert Vernon (1999). „Scalable Trigger Processing“. In: *Proceedings of the 15th International Conference on Data Engineering, 1999*. IEEE Computer Society, pp. 266–275.

Hendawi, Abdeltawab M., Jayant Gupta, Youying Shi, Hossam Fattah, and Mohamed H. Ali (2017). „The Microsoft Reactive Framework Meets the Internet of Moving Things“. In: *Proceedings of the 33rd IEEE International Conference on Data Engineering, ICDE 2017*. IEEE Computer Society, pp. 1150–1161.

Hendawi, Abdeltawab M., Youying Shi, Hossam Fattah, Jumana Karwa, and Mohamed H. Ali (2016). „RxSpatial: a framework for real-time spatio-temporal operations: demo“. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS 2016*. ACM, pp. 366–367.

Hewitt, Carl (2010). „Actor model of computation: scalable robust information systems“. In: *arXiv preprint arXiv:1008.1459*.

Hilbert, David (1935). „ "U about the continuous mapping of a line onto a surface ä chenst "u ck“. In: *Third volume: Analysis textperiodcentered Fundamentals of Mathematics textperiodcentered Physics Various*. springer, pp. 1–2.

Holl, Stephan and Hans Plum (2009). „PostGIS“. In: *GeoInformatics* 3, pp. 34–36.

Honkote, Vinayak, Dibyendu Ghosh, Karthik Narayanan, Ankit Gupta, and Anuradha Srinivasan (2020). „Design and Integration of a Distributed, Autonomous and Collaborative Multi-Robot System for Exploration in Unknown Environments“. In: *Proceedings of the 2020 IEEE/SICE International Symposium on System Integration, SII 2020*. IEEE, pp. 1232–1237.

Hu, Junyan, Parijat Bhowmick, Farshad Arvin, Alexander Lanzon, and Barry Lennox (2020). „Cooperative Control of Heterogeneous Connected Vehicle Platoons: An Adaptive Leader-Following Approach“. In: *IEEE Robotics Autom. Lett.* 5.2, pp. 977–984.

Hughes, James N, Andrew Annex, Christopher N Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest (2015). „Geomesa: a distributed architecture for spatio-temporal fusion“. In: *SPIE: Geospatial Informatics, Fusion, and Motion Video Analytics V*. Vol. 9473, 94730F.

IBM Food Trust: trust and transparency in our food (2018). `https://www.ibm.com/blockchain/solutions/food-trust`.

Iyer, Anand Padmanabha and Ion Stoica (2017). „A scalable distributed spatial index for the internet-of-things“. In: *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017*. ACM, pp. 548–560.

Jacobs, Steven, Xikui Wang, Michael J. Carey, Vassilis J. Tsotras, and Md. Yusuf Sarwar Uddin (2020). „BAD to the bone: Big Active Data at its core“. In: *Proceedings of the 46th international conference on Very large data bases, VLDB 2020* 29.6, pp. 1337–1364.

Jagadish, Hosagrahar V (1990). „Linear clustering of objects with multiple attributes“. In: *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pp. 332–342.

Javaid, Mohd and Ibrahim Haleem Khan (2021). „Internet of Things (IoT) enabled healthcare helps to take the challenges of COVID-19 Pandemic“. In: *Journal of Oral Biology and Craniofacial Research* 11.2, pp. 209–214.

Jensen, Christian S., Dan Lin, and Beng Chin Ooi (2004). „Query and Update Efficient B+-Tree Based Indexing of Moving Objects“. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004*. Morgan Kaufmann, pp. 768–779.

Jensen, Christian S. and Stardas Pakalnis (2007). „TRAX - Real-World Tracking of Moving Objects“. In: *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007*. ACM, pp. 1362–1365.

Jeppesen, Jacob Hoxbroe, Emad Ebeid, Rune Hylsberg Jacobsen, and Thomas Skjødeberg Toftegaard (2018). „Open geospatial infrastructure for data management and analytics in interdisciplinary research“. In: *Computers and Electronics in Agriculture* 145, pp. 130–141.

Ji, Yuxuan and Nikolas Geroliminis (2012). „On the spatial partitioning of urban transportation networks“. In: *Transportation Research Part B: Methodological* 46.10, pp. 1639–1656.

Jones, Christopher B. (1997). *Geographical Information Systems and Computer Cartography*. Prentice Hall.

Kambalyal, Channu (2010). „3-tier architecture“. In: *Retrieved On* 2, p. 34.

Karmani, Rajesh K and Gul Agha (2011). *Actors*. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.226.5767`.

Kazemitabar, Seyed Jalal, Ugur Demiryurek, Mohamed H. Ali, Afsin Akdogan, and Cyrus Shahabi (2010). „Geospatial Stream Query Processing using Microsoft SQL Server StreamInsight“. In: *Proceedings of the 36th International Conference on Very Large Data Bases, VLDB 2010* 3.2, pp. 1537–1540.

Khajenasiri, Iman, Abouzar Estebsari, Marian Verhelst, and Georges Gielen (2017). „A review on Internet of Things solutions for intelligent energy control in buildings for smart city applications“. In: *Energy Procedia* 111, pp. 770–779.

Kimball, Ralph and Margy Ross (2013). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. 3rd. Wiley Publishing.

Kondo, Andréia Akemi, Claudia Bauzer Medeiros, Evandro Bacarin, and Edmundo Roberto Mauro Madeira (2007). „Traceability in Food for Supply Chains.“ In: *Proceedings of the International Conference on Web Information Systems and Technologies, WEBIST 2007*, pp. 121–127.

Kraijak, Surapon and Panwit Tuwanut (2015). „A survey on IoT architectures, protocols, applications, security, privacy, real-world implementation and future trends“. In: *Proceedings of the 11th International Conference on Wireless Communications, Networking and Mobile Computing, WiCOM 2015*, pp. 1–6.

Kraska, Tim, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis (2018). „The Case for Learned Index Structures“. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD 2018*. ACM, pp. 489–504.

Lee, In and Kyoochun Lee (2015). „The Internet of Things (IoT): Applications, investments, and challenges for enterprises“. In: *Business Horizons* 58.4, pp. 431–440.

Li, Hao and Fawzi Nashashibi (2013). „Cooperative Multi-Vehicle Localization Using Split Covariance Intersection Filter“. In: *IEEE Intell. Transp. Syst. Mag.* 5.2, pp. 33–44.

Li, Pengfei, Hua Lu, Qian Zheng, Long Yang, and Gang Pan (2020a). „LISA: A Learned Index Structure for Spatial Data“. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD 2020*. ACM, pp. 2119–2133.

Li, Zhe, Tsz Nam Chan, Man Lung Yiu, and Christian S. Jensen (2020b). „PolyFit: Polynomial-based Indexing Approach for Fast Approximate Range Aggregate Queries“. In: *CoRR* abs/2003.08031. arXiv: 2003.08031.

Lindley, Craig A. and Mirjam Eladhari (2005). „Narrative Structure in Trans-Reality Role-Playing Games: Integrating Story Construction from Live Action,

Table Top and Computer-Based Role-Playing Games“. In: *Proceedings of the Digital Games Research Conference 2005, Changing Views: Worlds in Play*.

Lu, Jiamin and Ralf Hartmut Güting (2014). „Parallel SECONDO: A practical system for large-scale processing of moving objects“. In: *Proceedings of the IEEE 30th International Conference on Data Engineering, ICDE 2014*. IEEE Computer Society, pp. 1190–1193.

Luo, Chen and Michael J. Carey (2018). „Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems“. In: *CoRR* abs/1808.08896. arXiv: 1808.08896.

McLellan, Charles (2020). *Smart farming: How IoT, robotics, and AI are tackling one of the biggest problems of the century*. https://www.techrepublic.com/article/smart-farming-how-iot-robotics-and-ai-are-tackling-one-of-the-biggest-problems-of-the-century/.

Microsoft Reactive Framework - ReactiveX: An API for asynchronous programming with observable streams (2020). http://reactivex.io/.

Microsoft SQL Server Spatial (2020). https://gdal.org/drivers/vector/mssqlspatial.html.

Miller, Jeremiah, Miles Raymond, Josh Archer, Seid Adem, Leo Hansel, Sushma Konda, Malik Luti, Yao Zhao, Ankur Teredesai, and Mohamed H. Ali (2011). „An Extensibility Approach for Spatio-temporal Stream Processing Using Microsoft StreamInsight“. In: *Proceedings of the Advances in Spatial and Temporal Databases - 12th International Symposium, SSTD 2011*. Vol. 6849. Lecture Notes in Computer Science. Springer, pp. 496–501.

Mills, David L. (1992). „Network Time Protocol (Version 3) Specification, Implementation and Analysis“. In: *RFC* 1305, pp. 1–109.

Minoli, Daniel, Kazem Sohraby, and Benedict Occhiogrosso (2017). „IoT Security (IoTSec) Mechanisms for e-Health and Ambient Assisted Living Applications“. In: *Proceedings of the Second IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies, CHASE 2017*. Ed. by Paolo Bonato and Honggang Wang. IEEE, pp. 13–18.

Mitsakis, Evangelos, Areti Kotsi, and Vasileios Psonis (2020). „C-ITS bundling for integrated traffic management“. In: *CoRR* abs/2011.03425. arXiv: 2011.03425.

Momjian, Bruce (2001). *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

Morton, Guy M (1966). *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York.

Mousavi, A, M Sarhadi, A Lenk, and S Fawcett (2002). „Tracking and traceability in the meat processing industry: a solution“. In: *British Food Journal* 104.1, pp. 7–19.

Muthuramalingam, S, A Bharathi, N Gayathri, R Sathiyaraj, B Balamurugan, *et al.* (2019). „IoT based intelligent transportation system (IoT-ITS) for global perspective: A case study“. In: *Internet of Things and Big Data Analytics for Smart Generation*. Springer, pp. 279–300.

Nahrstedt, Klara, Hongyang Li, Phuong Nguyen, Siting Chang, and Long H. Vu (2016). „Internet of Mobile Things: Mobility-Driven Challenges, Designs and Implementations“. In: *Proceedings of the First IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*. IEEE Computer Society, pp. 25–36.

Nathan, Vikram, Jialin Ding, Mohammad Alizadeh, and Tim Kraska (2020). „Learning Multi-Dimensional Indexes“. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD 2020*. ACM, pp. 985–1000.

Newell, Andrew, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein (2016). „Optimizing distributed actor systems for dynamic interactive services“. In: *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016*. ACM, 38:1–38:15.

Nguyen, Phu H., Åsmund Hugo, Karl Svantorp, and Bjørn Magne Elnes (2020). „Towards a Simulation Framework for Edge-to-Cloud Orchestration in C-ITS“. In: *Proceedings of the 21st IEEE International Conference on Mobile Data Management, MDM 2020*. IEEE, pp. 354–358.

Ni, Daiheng (2016). „Traffic Flow Theory“. In: *Chapter* 24, pp. 361–377.

Nie, Yun-Feng, Hai-Ling Liu, and Hu Xu (2011). „Research on GeoWebCache tile map service middleware“. In: *Science of Surveying and Mapping* 36, pp. 207–209.

Nievergelt, Jürg, Hans Hinterberger, and Kenneth C. Sevcik (1984). „The Grid File: An Adaptable, Symmetric Multikey File Structure“. In: *ACM Trans. Database Syst.* 9.1, pp. 38–71.

Nishimura, Shoji, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi (2013). „MD-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services“. In: *Distributed Parallel Databases* 31.2, pp. 289–319.

OGC (2021). *OGC Standards*. `https://www.ogc.org/docs/is`.

Ojagh, Soroush, Mohammad Reza Malek, Sara Saeedi, and Steve Liang (2020). „A location-based orientation-aware recommender system using IoT smart

devices and Social Networks“. In: *Future Generation Computer Systems* 108, pp. 97–118.

Orbit (2020). https://www.orbit.cloud/orbit/.

Orenstein, Jack A. (1982). „Multidimensional Tries Used for Associative Searching“. In: *Inf. Process. Lett.* 14.4, pp. 150–157.

Who Is Using Orleans? (July 2018). http://dotnet.github.io/orleans/Community/Who-Is-Using-Orleans.html.

Cluster Management in Orleans (2020). https://dotnet.github.io/orleans/Documentation/implementation/cluster_management.html.

Orleans Microsoft - Microsoft Orleans Documentation (2021). https://dotnet.github.io/orleans/docs/index.html.

Orleans Silo - Microsoft Orleans Documentation (2020). https://dotnet.github.io/orleans/Documentation/clusters_and_clients/silo_lifecycle.html.

Microsoft (2021). *Microsoft Orleans Documentation - Orleans silo*. https://dotnet.github.io/orleans/docs/host/silo_lifecycle.html.

Orleans Streams - Microsoft Orleans Documentation (2020). https://dotnet.github.io/orleans/docs/streaming/index.html.

Grain Call Filters (Oct. 2018). https://dotnet.github.io/orleans/Documentation/grains/interceptors.html.

Owens, Mike and Grant Allen (2010). *SQLite*. Springer.

Pandey, Varun, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper (2016). „High-Performance Geospatial Analytics in HyPerSpace“. In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD2016*. ACM, pp. 2145–2148.

Papp, Zoltan, CHRIS Brown, and CHRISTINE Bartels (2008). „World modeling for cooperative intelligent vehicles“. In: *Proceedings of the 2008 IEEE Intelligent Vehicles Symposium*. IEEE, pp. 1050–1055.

Pasricha, Sudeep, Raid Ayoub, Michael Kishinevsky, Sumit K. Mandal, and Ümit Y. Ogras (2020). „A Survey on Energy Management for Mobile and IoT Devices“. In: *IEEE Des. Test* 37.5, pp. 7–24.

Paul, Somnath and TV Sarath (2018). „End to End IoT Based Hazard Monitoring System“. In: *Proceedings of the 2018 International Conference on Inventive Research in Computing Applications, ICIRCA 2018*. IEEE, pp. 106–110.

Pelanis, Mindaugas, Simonas Saltenis, and Christian S. Jensen (2006). „Indexing the past, present, and anticipated future positions of moving objects“. In: *ACM Trans. Database Syst.* 31.1, pp. 255–298.

Persson, Per and Ola Angelsmark (2015). „Calvin–merging cloud and iot“. In: *Procedia Computer Science* 52, pp. 210–217.

The Ptolemy Project: Accessors (Oct. 2018). https://ptolemy.berkeley.edu/accessors/.

Qin, Lele, Shuang Feng, and Hongyi Zhu (2018). „Research on the technological architectural design of geological hazard monitoring and rescue-after-disaster system based on cloud computing and Internet of things“. In: *Int. J. Syst. Assur. Eng. Manag.* 9.3, pp. 684–695.

Reactive Extension (2020). https://docs.microsoft.com/en-us/previous-versions/dotnet/reactive-extensions/hh242985(v=vs.103)?redirectedfrom=MSDN.

Reactive Streams (2021). https://www.reactive-streams.org/.

Reactor - Create Efficient Reactive Systems (2021). https://projectreactor.io/.

Rigaux, Philippe, Michel Scholl, and Agnès Voisard (2002). *Spatial databases - with applications to GIS*. Elsevier.

Rong, Yi and Christos Faloutsos (1991). *Analysis of the clustering property of Peano curves*. University of Maryland.

Sagan, Hans (2012). *Space-filling curves*. Springer Science & Business Media.

Salvaneschi, Guido and Mira Mezini (2013). „Towards Reactive Programming for Object-Oriented Applications“. In: *LNCS Trans. Aspect Oriented Softw. Dev.* 11, pp. 227–261.

Sánchez, Daniel Díaz, R. Simon Sherratt, Patricia Arias, Florina Almenárez, and Andrés Marín (2015). „Enabling actor model for crowd sensing and IoT“. In: *Proceedings of the International Symposium on Consumer Electronics, ISCE 2015*. IEEE, pp. 1–2.

Sarwat, Mohamed, Sameh Elnikety, Yuxiong He, and Gabriel Kliot (2012). „Horton: Online Query Execution Engine for Large Distributed Graphs“. In: *Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012*. IEEE Computer Society, pp. 1289–1292.

SEGES Landbrug & Fødevarer F.m.b.A. website (2018). https://www.seges.dk/en.

Sellis, Timos, Nick Roussopoulos, and Christos Faloutsos (1987). *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*. Tech. rep.

SenMoS: your sensor monitoring system (2018). https://senmos.dk/.

Shah, Vivek and Marcos Antonio Vaz Salles (2018a). „Reactors: A Case for Predictable, Virtualized Actor Database Systems“. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD 2018*. Ed.

by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, pp. 259–274.

Shah, Vivek and Marcos Vaz Salles (2018b). „Actor-Relational Database Systems: A Manifesto“. In: *CoRR* abs/1707.06507.

Shaikh, Salman Ahmed, Komal Mariam, Hiroyuki Kitagawa, and Kyoung-Sook Kim (2020). „GeoFlink: A Distributed and Scalable Framework for the Real-time Processing of Spatial Streams“. In: *Proceedings of the 29th ACM International Conference on Information and Knowledge Management, CIKM 2020*. ACM, pp. 3149–3156.

Sharma, Vishal, Ilsun You, Karl Andersson, Francesco Palmieri, Mubashir Husain Rehmani, and Jae-Deok Lim (2020). „Security, Privacy and Trust for Smart Mobile- Internet of Things (M-IoT): A Survey“. In: *IEEE Access* 8, pp. 167123–167163.

Shen, Zhitao, Vikram Kumaran, Michael J Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche, and Kim Macpherson (2015). „CSA: Streaming Engine for Internet of Things.“ In: *IEEE Data Eng. Bull.* 38.4, pp. 39–50.

Shi, Youying, Abdeltawab M. Hendawi, Hossam Fattah, and Mohamed H. Ali (2016a). „RxSpatial: Reactive Spatial Library for Real-Time Location Tracking and Processing“. In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016,* ACM, pp. 2165–2168.

Shi, Youying, Abdeltawab M. Hendawi, Jayant Gupta, Hossam Fattah, and Mohamed H. Ali (2016b). „RxSpatial: the reactive spatial library“. In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016*. ACM, 64:1–64:4.

Shorinwa, Ola, Javier Yu, Trevor Halsted, Alex Koufos, and Mac Schwager (2020). „Distributed Multi-Target Tracking for Autonomous Vehicle Fleets“. In: *Proceedings of the 2020 IEEE International Conference on Robotics and Automation, ICRA 2020*. IEEE, pp. 3495–3501.

Sidlauskas, Darius and Christian S. Jensen (2014). „Spatial Joins in Main Memory: Implementation Matters!“ In: *Proc. VLDB Endow.* 8.1, pp. 97–100.

Sidlauskas, Darius, Simonas Saltenis, Christian W. Christiansen, Jan M. Johansen, and Donatas Saulys (2009). „Trees or grids?: indexing moving objects in main memory“. In: *Proceedings of the 17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009*. ACM, pp. 236–245.

Šidlauskas, Darius, Simonas Šaltenis, and Christian S Jensen (2012). „Parallel main-memory indexing for moving-object query and update workloads“. In:

Proceedings of the 2012 ACM SIGMOD international conference on management of data, pp. 37–48.

Sidlauskas, Darius, Simonas Saltenis, and Christian S. Jensen (2014). „Processing of extreme moving-object update and query workloads in main memory". In: *Proceedings of the 40th international conference on Very large data bases, VLDB 2014* 23.5, pp. 817–841.

Singhal, Ayush, Rakesh Pant, and Pradeep Sinha (2018). „AlertMix: A Big Data platform for multi-source streaming data". In: *arXiv preprint arXiv:1806.10037*.

SingleStore: All Data, One Platform. (2021). `https://www.singlestore.com/`.

Sørensen, Claus G and Dionysis D Bochtis (2010). „Conceptual model of fleet management in agriculture". In: *Biosystems Engineering* 105.1, pp. 41–50.

Sowell, Benjamin, Marcos Antonio Vaz Salles, Tuan Cao, Alan J. Demers, and Johannes Gehrke (2013). „An Experimental Analysis of Iterated Spatial Joins in Main Memory". In: *Proceedings of the 39th international conference on Very large data bases, VLDB 2013* 6.14, pp. 1882–1893.

Srinivasan, CR, B Rajesh, P Saikalyan, K Premsagar, and Eadala Sarath Yadav (2019). „A review on the different types of Internet of Things (IoT)". In: *Journal of Advanced Research in Dynamical and Control Systems* 11.1, pp. 154–158.

Stonebraker, Michael, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland (2007). „The End of an Architectural Era (It's Time for a Complete Rewrite)". In: *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007*. ACM, pp. 1150–1160.

Facts and History (2018). `https://www.storebaelt.dk/english/bridge`.

Microsoft StreamInsight (2020). `https://www.microsoft.com/en-us/download/search.aspx?q=StreamInsight`.

Taft, Rebecca (2017). „Elastic database systems". PhD thesis. Massachusetts Institute of Technology, Cambridge, USA.

Tak, Sehyun, Jinsu Yoon, Soomin Woo, and Hwasoo Yeo (2020). „Sectional information-based collision warning system using roadside unit aggregated connected-vehicle information for a cooperative intelligent transport system". In: *Journal of advanced transportation* 2020.

Tang, Chuzhe, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen (2020). „XIndex: a scalable learned index for multicore data storage". In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2020*. ACM, pp. 308–320.

Tang, MingJie, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref (2016). „LocationSpark: A Distributed In-Memory Data Management

System for Big Spatial Data“. In: *Proceedings of the 42nd international conference on Very large data bases, VLDB 2016* 9.13, pp. 1565–1568.

Teorey, Toby J (1999). *Database modeling & design*. Morgan Kaufmann.

The Reactive Relational Database Connectivity (R2DBC) (2021). `https://r2dbc.io/`.

Akka Migration Guide 2.3.x to 2.4.x (2018). `https://doc.akka.io/docs/akka/2.4/project/migration-guide-2.3.x-2.4.x.html`.

Tundis, Andrea, Humayun Kaleem, and Max Mühlhäuser (2020). „Detecting and Tracking Criminals in the Real World through an IoT-Based System“. In: *Sensors* 20.13, p. 3795.

Uhlemann, E. (2018). „Time for Autonomous Vehicles to Connect [Connected Vehicles]“. In: *IEEE Vehicular Technology Magazine* 13.3, pp. 10–13.

OMG Unified Modeling Language (OMG UML) Version 2.5.1 (Nov. 2018). `https://www.omg.org/spec/UML/2.5.1/PDF`.

Universe, The Parallel (2012). *Introducing Spacebase: a New Realtime Spatial Data Store*.

Vo, Hoang, Yanhui Liang, Jun Kong, and Fusheng Wang (2018). „iSPEED: a scalable and distributed in-memory based spatial query system for large and structurally complex 3D data“. In: *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*. Vol. 11. 12. NIH Public Access, p. 2078.

Vu, Tin, Alberto Belussi, Sara Migliorini, and Ahmed Eldawy (2020). „Using Deep Learning for Big Spatial Data Partitioning“. In: *ACM Trans. Spatial Algorithms Syst.* 7.1, 3:1–3:37.

Vu, Tin and Ahmed Eldawy (2020). „R*-Grove: Balanced Spatial Partitioning for Large-Scale Datasets“. In: *Frontiers Big Data* 3, p. 28.

Wang, Haixin, Xiaoyi Fu, Jianliang Xu, and Hua Lu (2019a). „Learned Index for Spatial Queries“. In: *Proceedings of the 20th IEEE International Conference on Mobile Data Management, MDM 2019*. IEEE, pp. 569–574.

Wang, Yiwen (2018). „Vecstra: An Efficient and Scalable Geospatial In-Memory Cache.“ In: *PhD@ VLDB*.

Wang, Yiwen, Júlio César dos Reis, Kasper Myrtue Borggren, Marcos Antonio Vaz Salles, Claudia Bauzer Medeiros, and Yongluan Zhou (2019b). „Modeling and Building IoT Data Platforms with Actor-Oriented Databases“. In: *Proceedings of Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019*. OpenProceedings.org, pp. 512–523.

Widom, Jennifer and Stefano Ceri (1996). *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann.

Wikipedia (2020). *Speed Limits by country*. `https://en.wikipedia.org/wiki/Speed_limits_by_country`.

Wolfson, Ouri, Bo Xu, Sam Chamberlain, and Liqin Jiang (1998). „Moving Objects Databases: Issues and Solutions“. In: *Proceedings of the 10th International Conference on Scientific and Statistical Database Management, 1998*. IEEE Computer Society, pp. 111–122.

Woods, Orlando (2021). „Experiencing the unfamiliar through mobile gameplay: Pokémon go as augmented tourism“. In: *Area* 53.1, pp. 183–190.

Wu, Yingjun, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber (2019). „Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations“. In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD 2019*. ACM, pp. 1223–1240.

Xie, Dong, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo (2016). „Simba: Efficient In-Memory Spatial Analytics“. In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016*. ACM, pp. 1071–1085.

You, Simin, Jianting Zhang, and Le Gruenwald (2015). „Large-scale spatial join query processing in Cloud“. In: *Proceedings of the 31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015*. IEEE Computer Society, pp. 34–41.

Yu, Jia, Jinxuan Wu, and Mohamed Sarwat (2015). „GeoSpark: a cluster computing framework for processing large-scale spatial data“. In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2015*. ACM, 70:1–70:4.

Yu, Jia, Jinxuan Wu, and Mohamed Sarwat (2016). „A demonstration of GeoSpark: A cluster computing framework for processing big spatial data“. In: *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, pp. 1410–1413.

Zhou, Muyi (2020). *San Francisco Mobility Trend Report 2018*. `https://www.sfmta.com/reports/san-francisco-mobility-trends-report-2018`.

Zhou, Xiaofang, David J. Abel, and David Truffet (1998). „Data Partitioning for Parallel Spatial Join Processing“. In: *GeoInformatica* 2.2, pp. 175–204.

Zou, Tao, Guozhang Wang, Marcos Antonio Vaz Salles, David Bindel, Alan J. Demers, Johannes Gehrke, and Walker M. White (2011). „Making time-stepped applications tick in the cloud“. In: *Proceedings of the ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC 2011*. ACM, p. 20.