CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Query Processing on Spatio-Temporal Data Streams from Moving Objects

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

vorgelegt von

Tobias Leo Brandt

Gutachter:

apl. Prof. Dr.-Ing. Jürgen Sauer
Prof. Dr. Thomas Brinkhoff

Tag der Disputation: 11. November 2019

Zusammenfassung

Die zunehmende Verfügbarkeit von mobilen Datenverbindungen, die Miniaturisierung von GPS-Empfängern und der gestiegene Bedarf an dem Austausch von ortsbezogenen Informationen lassen die Menge von Daten mit zusätzlichen spatio-temporalen Anteilen steigen. Gleichzeitig ist für viele Anwendungen die Zeit vom Eintreffen von Informationen bis zum Nutzen dieser durch geeignete Verarbeitung entscheidend. Gerade bei Daten von sich bewegenden Objekten, wie etwa regelmäßigen Positionsmeldungen von Fußgängern, Autos oder Schiffen entstehen besondere Herausforderungen bei der Verarbeitung von kontinuierlichen Datenströmen. Im maritimen Bereich ist die Verwendung solcher Daten bereits üblich, da die meisten Schiffe verpflichtet sind, ihre Position regelmäßig über das Automatic Identification System (AIS) zu melden.

Typische kontinuierliche Anfragen auf diesen Datenströmen könnten die Distanz zwischen sich bewegenden Objekten berechnen, um zum Beispiel eine Warnung zu erzeugen wenn sich Objekte zu nah kommen oder sich in unerlaubte Gebiete begeben. Hierbei muss berücksichtigt werden, dass nicht zu jedem Zeitpunkt die Position von jedem Objekt bekannt ist. Ein Lösungsansatz ist die Prädiktion von spatialen Informationen zu bestimmten Zeitpunkten.

Datenstrommanagementsysteme (DSMS) stellen eine geeignete Grundlage für die Verarbeitung von kontinuierlichen Anfragen auf Datenströmen dar. Sie bieten Lösungen für typische Herausforderungen im Datenstromumfeld, wie etwa Fensteransätze zur Begrenzung eines potentiell unendlichen Stroms, und vermindern damit notwendige und aufwendige Neuentwicklungen im Umgang mit Datentrömen. Auch sind DSMS sehr flexibel und können somit für unterschiedliche Nutzungsszenarien eingesetzt werden, was sie von einer spezifischen Lösung für nur einen Anwendungsfall unterscheidet. Jedoch bedürfen die speziellen Anforderungen an kontinuierliche Anfragen auf Daten von sich bewegenden Objekten neuer Lösungsansätze. Insbesondere die zeitlich korrekte Verarbeitung bei nicht regelmäßig zur Verfügung stehenden Ortsinformationen erfordert eine Erweiterung von typischen Datenstromtechniken.

In dieser Arbeit wird ein Konzept zur Integration von Techniken zur Verarbeitung von Datenströmen sich bewegender Objekte in DSMS vorgestellt. Dies baut auf den theoretischen Grundlagen der Moving Object Algebra [GBE+00] sowie der Datenstromverarbeitung im Intervallansatz [KS09] auf. Durch die Verbindung der gegebenen Grundlagen dieser Arbeiten entsteht ein klares semantisches Fundament für bitemporale Datenstromanfragen, also Anfragen mit zwei zeitlichen Dimensionen, mit denen Anfragen sehr flexibel gestaltet werden können.

Diese Integration einer zweiten temporalen Dimension in den Kern eines DSMS neben das durch den Intervallansatz bereits bestehende Zeitintervall des Datenstroms ist eines der grundlegenden Konzepte dieser Arbeit. So können Attribute in Datenstromelementen selbst unterschiedliche Werte zu unterschiedlichen Zeitpunkten in dieser zweiten

Zeitdimension haben und damit einen Wert zu einem fast beliebig wählbaren Zeitpunkt vorhersagen.

Die Evaluation beinhaltet eine Implementierung der entwickelten Konzepte in das Data Stream Management System (DSMS) Framework Odysseus. Damit wird die Umsetzbarkeit des Konzeptes gezeigt. In der Evaluation zeigt sich, dass es möglich ist, das erstellte Konzept in ein existierendes System zu integrieren und dabei auch existierende Funktionalität wiederzuverwenden, sodass bestehende Fähigkeiten des Systems nicht neu implementiert werden müssen. Damit entsteht ein sehr flexibles System zur Umsetzung vieler Szenarien im Umfeld von sich bewegenden Objekten als auch darüber hinaus.

Die Leistungsfähigkeit und Funktionalität des entwickelten Systems wurden mit AIS Daten von der US-amerikanischen Küste anhand verschiedener spatio-temporaler Anfragen evaluiert. Die Ergebnisse zeigen, dass Anfragen, die dem ebenfalls entwickelten Konzept für spatio-temporale Anfragen auf Datenströmen von sich bewegenden Objekten folgen, einige Parameter besitzen, die die Performanz maßgeblich beeinflussen können. Ebenso zeigt die Evaluation, dass die Anfragen unter den evaluierten Voraussetzungen schneller als Echtzeit laufen und somit für diese Szenarien einsetzbar sind. Um die Performanz zu erhöhen, wurden außerdem spatio-temporale Filter-Techniken entwickelt, die die maßgeblichen Messwerte Latenz und Durchsatz signifikant erhöhen können und die Qualität der Ergebnisse dabei nur geringfügig beeinflussen. Außerdem zeigt die Evaluation, dass der entwickelte bitemporale Ansatz auch über die Szenarien und Domäne hinaus einsetzbar ist, selbst bei Daten, die keinen Ortsbezug besitzen.

Abstract

Due to the ubiquitous availability of mobile data connections, smaller and cheaper GPS receivers and the demand to share location enriched information, the amount of spatio-temporal data is increasing. Additionally, for many applications, a short latency between data arrival and query results is crucial to be able to work with the data immediately. Especially for data from moving objects, such as continuous location updates from pedestrians, cars or vessels, new challenges for the processing of continuous data streams arise. Within the maritime domain, the usage of such data streams is already common because most vessels are obligated to broadcast their location regularly via the Automatic Identification System (AIS).

Queries on these data streams could, for example, calculate the distance between these moving objects in a continuous manner and create warnings when objects are too close to each other or enter forbidden areas. Doing this, it has to be taken into account that the locations are not available for each point in time. One solution to deal with this is the prediction of locations for certain points in time.

Data Stream Management Systems (DSMS) offer a foundation for the processing of continuous queries on data streams. They implement solutions for typical challenges in the data stream environment, for example window approaches to limit the potentially infinite data stream, and therefore reduce the need for expensive development when working with data streams. Additionally, DSMS are very flexible and can be applied in various use cases, which is another difference to custom solutions for a specific use case. Nevertheless, DSMS lack support for the special requirements for data streams from moving objects. Especially the temporal-wise correct processing for data that is not available at all times requires an extension of existing data stream processing techniques.

In this thesis, a concept for the integration of techniques into DSMS for the processing of moving object data streams is proposed. It integrates the ideas of the Moving Object Algebra [GBE+00] with the interval approach for data stream processing [KS09]. Due to the connection of these two foundations a clear semantics for bitemporal queries is created, with which queries can be designed very flexible. The bitemporal stream is a main concept of this work. It describes the integration of a second temporal dimension next to the stream time interval into the core of the query processing. Doing that, attributes within a data stream element can have different values at different points in time and can therefore be predicted to an (partly) arbitrary point in time.

The evaluation is done by implementing the developed concept into the DSMS framework Odysseus to evaluate the feasibility of this approach. The implementation has shown that it is possible to integrate the concept into an existing DSMS an to reuse the existing functionality, resulting in a very flexible system without the need to reimplement existing features.

The performance and functional capabilities were evaluated with different spatio-temporal queries with real-world AIS data from the US coast. The results show that a query that follows the developed concept for spatio-temporal queries on moving object data streams has some parameters that can influence the performance significantly, but also that, for the evaluation scenario, the queries run faster than real-time. To improve the performance, spatio-temporal streaming filter algorithms have been developed and the evaluation shows that those can significantly improve the important query performance indicators latency and data rate with only a very limited amount of missed results. The evaluation further shows that the developed approach can be applied to a wider field, even in scenarios without spatio-temporal data.

Acknowledgments

Finishing a doctoral thesis, closing the last chapter and accomplishing the disputation as the final act of a years long journey through a topic so narrow that only few people around you know what you are thinking about is a little like coming home after a long trip through the world (of science). You leave something behind, you collected a lot of inspiration and you grew with many new challenges. Now, having finished this interesting part of my scientific career I want to say thank you to all of those who have accompanied me in the recent years.

The fruitful discussions with my former colleagues in the information systems group, sometimes late in the evening, were creative, fun and also gave inspirations for new ways to solve problems in my research. I already miss the late talks and sketching ideas on our whiteboards with Michael Brand and Cornelius Ludmann. Without a doubt the most and longest discussions I had were with my supervisor Marco Grawunder, whose door was always open for me, who had great hints to the right direction when I struggled with tricky problems, with whom I could discuss about tiny details as well as the great picture and who always helped me to find my way through the scientific landscape of journals and conferences. Thank you!

Furthermore, I want to thank my reviewers Jürgen Sauer and Thomas Brinkhoff, who helped me with advice in their respective domains and their critical questions during the process of developing the ideas of this thesis. I also want to thank Axel Hahn for being the chair of the thesis committee and for making the funding for this PhD project possible with the Safe Automation of Maritime Systems (SAMS) graduate school.

My PhD journey would not have been so much fun without the breaks at the nice places in life with my friends and family. A big “Thank you” to my closest friends who not only gave me the opportunity to not think about data streams all day, but actively pushed me to take breaks when needed so that I never lost the motivation and joy over my work.

The last paragraph is reserved for those who accompanied me the longest, who supported and believed in me the most. I want to thank my family with all my heart! Your support has been and still is infinitely valuable.

Thank you all for being on this journey with me! It has been a great time!

Tobias Brandt
Oldenburg, December 10^{th}, 2019

Contents

1 Introduction

Today it is taken for granted that the exact location on the earth can be determined with ease. Small and mobile Global Navigation Satellite System (GNSS) receivers [SRC+16] are cheap and broadly available. Together with mobile data connections, the locations of objects can be communicated to central services or shared with other vehicles. Smartphones, cars and vessels, for example, can be seen as moving objects which locations can be shared and analyzed. These continuous data streams of location information can be used to monitor and optimize traffic, offer Location Based Services (LBSs) and improve safety. For these applications, where location information is more important than ever, streaming spatio-temporal data needs to be managed and processed.

1.1 Motivation and Goals

Processing spatio-temporal data streams is done for diverse use cases [BG18]. LBSs can find crowded places [WKK+14], car navigation systems estimate the current traffic flow on streets [THC+14] and Maritime Vessel Traffic Services (VTSs) can improve safety and traffic efficiency on seas [PAA+17]. These use cases have some common requirements that introduce new challenges to the spatio-temporal data processing.

All of the use cases mentioned above have in common that near real-time updates of continuous queries are necessary. Traditional spatio-temporal data management is often done with Data Base Management Systems (DBMS) [GAA+05, AR99, ESRa, EGSV99, KSF+03] or, more recently, with spatio-temporal key-value-stores from the emerging NoSQL field such as GeoMesa [FEHL13] as well as in MapReduce environments such as ST-Hadoop [Ala17]. Nevertheless, these systems lack the support for continuous queries with near real-time results and are therefore not optimal for use cases which require continuous, event-driven results [Bra17]. Data Stream Management Systems (DSMS) [CCD+03, AAB+05, ÇAA+16, KS04b, AGG+12] in contrast are designed to process data streams and already support such queries. Therefore, they are a useful foundation for continuous spatio-temporal data streams [BG17, Bra17].

These spatio-temporal data streams are generated by different and diverse sources—to process them, one needs to fuse different streams to query them together. It is possible that data streams need to be enriched with static data, e. g., if a current location of a vessel needs to be combined with the known maximum speed of it. Queries on streams and the data sources for these can change over time. For example, new data streams with information about other spatial areas can be added.

When creating applications that use spatio-temporal data streams, these characteristics need to be considered. If a completely new solution is implemented for every application, it is difficult to reuse existing solutions for common problems in the field of data streams and spatial processing. Additionally, the solutions would not be as flexible as

necessary for the diverse data sources and changing queries. DSMS in contrast are flexible platforms which already address common challenges for data stream processing, such as query optimization, distribution, time handling and implementation of standard operators and query languages [AGG+12]. Using it for these applications increases the solutions flexibility and ease of maintenance compared to custom implementations for every use case [BAG+12, Lud15, Lud17, BGA16, BBC+15].

Therefore, the goal of this work is to integrate spatio-temporal query processing into DSMS so that spatio-temporal queries for moving objects can be developed and executed in a flexible and efficient manner.

1.2 Problem Statement

When using a DSMS for spatio-temporal queries from moving objects, a couple of new challenges arise.

Spatio-temporal Operations Querying data from moving objects requires spatio-temporal operations on the data streams [HZEF16]. Such operations consider both spatial as well as temporal information. In other words, when doing a spatial query, the temporal information about the data needs to be considered, too. For example, the spatial query "Which objects are close to object X?" is extended by a temporal aspect, e. g., "Which objects are close to object X now?".

A challenge for these operations is that the spatial information is not available in a temporal synchronized manner. For different sources, in this case different moving objects, the spatial measurement, i. e., the location, is only available at different points in time [BAG+12]. In other words: for a certain point in time, spatial information is only available for a number of moving objects (or none), not for all moving objects. Nevertheless, the lack of a location information for a certain moving object does not mean that the location does not need to be taken into account. Therefore, a solution to solve this discrepancy needs to be found, e. g., via integrating prediction into the query processing.

Considering Efficiency Many spatio-temporal use cases demand near real-time results and cannot afford high latencies between new data and new results (e. g., [HZEF16, TMRDR12, CZC+13, PAA+17]). Therefore, the design of the DSMS extension needs to take efficiency into account. This is, for example, relevant for windows. When querying data streams, the concept of windows is often used to limit the data stream to a certain range in the past. Some data of the stream is kept in memory for a certain amount of time and is discarded afterwards. Operations, e. g., aggregations, can work on the data of the window. When working with spatio-temporal data, index structures for efficient access to the data can possibly help to improve efficiency for the overall query. Nevertheless, such an index structure would need to consider the characteristics of data streams, e. g., high fluctuation.

Compatibility to Existing DSMS Concepts The motivation of this work is to combine the features of a DSMS with new spatio-temporal capabilities to process moving object data streams. The newly added parts should work seamlessly with the existing concepts so that standard operations of a DSMS can be reused. Among these existing concepts is the time model (how to annotate the validity time of a data element), operators (such as select, join, etc.) and data types (e. g., by using a spatial data type in a way that new and old operators can work with it).

Research Question

From these challenges, the following research question arises:

How can queries on spatio-temporal data streams from moving objects be expressed flexibly and with generic semantics and processed efficiently?

The question consists of two parts: the first part is about the flexibility and semantics of queries and the second part about the efficiency. As explained above, the advantage of a DSMS over a custom solution for a use case is the possibility to adapt queries for multiple use cases and new situations easily. Therefore, the extension of a DSMS to work with spatio-temporal data streams should also be flexible and not bound to one use case. A generic semantics, a more formal framework to define such queries, helps with that.

When running continuous, event-driven queries and expecting near real-time results, the efficiency is of high importance. DSMS already consider efficiency for this purpose, for example, by using in-memory processing. Spatio-temporal queries propose new challenges in this field, for example, when joining huge amounts of moving objects to calculate close neighbors. This part of the research question asks if and how these challenges can be tackled in a streaming environment, which is heavily different from traditional spatial database environments.

1.3 Research Method

For the research of this dissertation, the design science research methodology from [PTRC07] is used. It is a process for design science research in information systems. [HMPR04] provide seven guidelines for design science research in information systems. According to them, “the research must provide an artifact created to address a problem” [PTRC07]. The research is the search for a solution for a problem with existing knowledge. The artifacts contain all parts of the solution, for example, a software product. The artifacts need to be evaluated for quality and the results need to be published to the scientific community [PTRC07].

[PTRC07] provide a process to follow these rules. It is depicted in Figure 1.1. The iterative process has six steps and four possible entry points. The first step is the problem

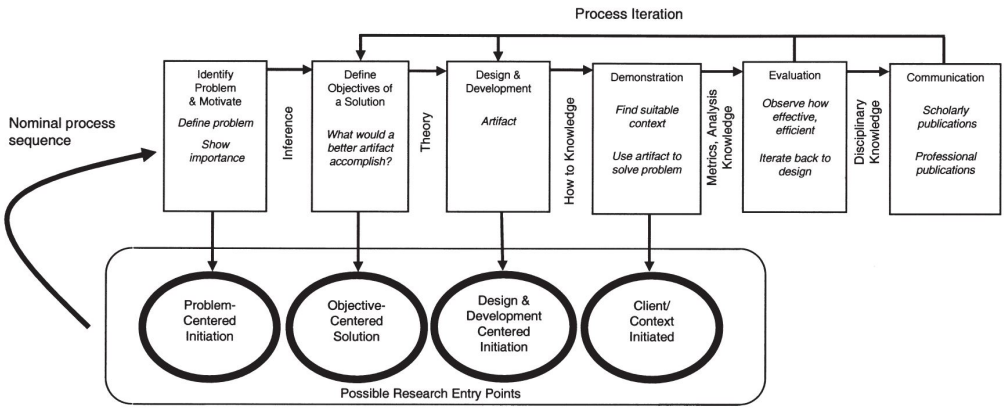


Figure 1.1: Process model of the Design Science Research Method by [PTRC07]

identification. It defines a problem and also shows the importance, e. g., by describing the practical use of a solution. The problem definition is used to define the objectives of the solution in the second step. These are later used to measure the qualitative and quantitative characteristics of the artifact. Based on the theory, an artifact is created in the design and development step. An artifact can be, for example, a model, a method or an instantiation, which includes software products.

In the next step, the artifact can be used to demonstrate how to solve the defined problem. This can be done by simulations, experiments or other appropriate activities [PTRC07]. Based on the demonstration, the artifact is evaluated against the objectives from the second step. The form of evaluation depends on the objectives and can include “system performance, such as response time or availability” [PTRC07]. If the objectives are not met, the process can be iterated through again to change the objectives or the creation of the artifact. The last step is the communication, for example, with scientific publication. The feedback from this step can also lead to another iteration [PTRC07].

The artifact in this thesis is mainly a concept on how to integrate moving object stream processing into a DSMS and a software that implements this concept for demonstration and evaluation purposes. The communication is done via scientific publications in the computer science community including this thesis.

1.4 Organization of the Thesis

The thesis starts with an introduction into the technical foundations of this work in Chapter 2. Especially the characteristics of data streams from moving objects and the basics of Data Stream Management Systems and spatial query processing are explained. Related work is also discussed in this chapter.

Chapter 3 and 4 contain the main contribution of this work. They describe the integration of spatio-temporal query processing on data streams from moving objects into a DSMS. The design and integration is split into two parts: the logical integration and the physical integration. This separation follows the differentiation into a logical and a physical part, which is inspired by [Krä07] as one of the main foundations of this thesis. Chapter 3 contains the logical integration and lays the theoretical foundations for this work. Based on the developed definitions in this chapter, Chapter 4 describes the physical integration, which is closer to an implementation and involves solutions to more practical problems that occur with potentially unbounded data streams and data from moving objects. It also addresses the second part of the research question and discusses possibilities and limitations of a more efficient query processing on this kind of data.

Even though the physical integration is closer to an implementation, the implementation itself rises some interesting questions and has some additional practical challenges that are explained in Chapter 5. The foundation for this chapter is an existing and extendable open-source DSMS named Odysseus. The sections describe the necessary extensions and changes to the system and gives some first examples on how the new possibilities can be used. Finally, Chapter 6 showcases the added capabilities with scenarios on real and artificial data streams and discusses limitations of the solution. Next to the evaluation of the flexibility with showcases of queries, the performance of the system is evaluated to review the second part of the research question and to identify the limitations of this approach.

Finally, Chapter 7 concludes this work, summarizes the added value and capabilities to a DSMS for moving object data streams and gives an outlook to future work that can be done in this field.

2 Background

This chapter provides basic knowledge about important topics of this thesis. First, the characteristics of the data, the starting point of the data processing, is explained in Section 2.1. In addition to the aspects of data streams, the aspects of spatial and temporal data are important as well. These factors define the way the data can be processed and lead to Data Stream Management Systems in Section 2.2 as a foundation to query data streams. Section 2.3 defines logical and physical data streams and explains the special characteristics of moving object data streams. Section 2.4 describes relevant aspects of spatial data processing as another foundation of this work. Combining spatial and temporal data processing, Section 2.5 explains the concept of the moving object algebra.

Vessel Traffic Services are a possible application for this work. To give the reader a practical example of how the output of this thesis could be used, VTSs are explained in Section 2.6. Related work is discussed in Section 2.7 and takes a look at Moving Object Databases and other streaming approaches for moving object data.

2.1 Moving Object Data Streams

A moving object data stream is generated from moving objects which actively report their location. It has three important attributes: (1) it is a data stream, (2) the data elements have spatial and (3) temporal information [BG17, BG18].

Data Stream

Data streams are typically generated by active data sources, i. e., the data sources push their data to the receivers[1]. This push based approach is one of the distinguishing characteristics of data streams that are different when compared to other similar data sources such as time series data [KS09, Gei13, GBKM14, Gal16].

- **Active data sources** The data sources push their data to the receivers. The receivers do not need to pull it from the sources but must handle the incoming data in the data rate that the source defines. Such a data source could be, in the context of moving objects, a smartphone that actively sends its current location to a LBS or a vessel that regularly sends its location information via the Automatic Identification System (AIS).
- **Unbounded** A data stream is potentially unbounded. It is not known in advance if and when the stream will end. Therefore, it is not feasible to wait for the stream to end to query the content afterwards. Another consequence is that it is infeasible to store the whole stream as the amount of data is potentially unbounded, too. Again,

[1] It is also possible that the DSMS pulls the data from the sources if the source is not able to push the data. In that case, the data is processed push-based after it was pulled by the DSMS.

using the AIS data stream as an example, the stream of messages will not stop as it is required to regularly broadcast the locations.

- **No control** The receiver of the data stream has no control over it, hence, it cannot change the data rate, accuracy or other characteristics of the stream. Receiving an AIS stream, the receiver cannot change the data rate of the location updates.
- **Only once** A data element in the stream is only send once and there is no possibility to request an element again. For example, an AIS receiver cannot ask a vessel to send its last or other previous messages again. The data elements have to be directly handled by the receiving system.

Spatial Information

A spatio-temporal data stream from moving objects contains spatial information. In this case it is mostly the location of the moving object as a single coordinate that references to the earth. Coordinate systems can be separated into two common types, namely Global Coordinate Systems (GCSs) and Projected Coordinate Systems (PCSs) [ESRb]. A very common GCS is the World Geodetic System 1984 (WGS 84) [LGMR10], a common PCS is Universal Transverse Mercator (UTM) [ESRb]. The location is typically measured with GNSS, the most common example is the Global Positioning System (GPS) [LGMR10]. It is also possible to use local coordinate systems as opposed to global coordinate systems. For example, this can be used to determine the location of a player on a sports field with the corners of the field as the reference points [BBC+15, GFW+11]. Nevertheless, this work focuses on data that is referenced to the earth's surface, mostly with WGS 84 as the GCS. Even though the examples are for objects on the earth, the concepts stay generic enough to work with other references as well.

Next to the bare location, the spatial information typically contains other information as well or is itself an additional information for other data, depending on the perspective. Moving objects can often be simplified to a point with no shape, so-called moving point objects [AG05]. Additionally, information such as the direction the object is facing can be part of the spatial information (e. g., within data about vessels, typically referred to as the Course Over Ground (COG) [PVB13]). Moving regions, i. e., moving objects with a shape and evolving regions, i. e., moving regions which shapes can change over time, are extensions to moving objects [MFBM14]. These data models can for example be used for weather phenomena such as hurricanes [WNPL13, WLN14, JG10]. Nevertheless, this work focuses on moving objects with no shape, as moving objects such as vessels can be simplified to points for many use cases [PVB13, PAA+17, GSF11, FXX+13, GLW08].

Temporal Information

Data streams are ordered by the time of the data stream elements [KS09, Gal16]. The time is measured by the data source at the time of capturing the data or by the receiving

system at the time of receiving the data [Gal16]. It can be attached to the data as a timestamp [KS09]. In case of moving object data, the time typically defines when the object has been at a certain location.

2.1.1 Restricted and Unrestricted Movement

Moving objects can move freely in space or can be restricted to a certain network. For example, trains and cars are typically restricted to their railway and road networks and cannot move freely in space. The network can be represented as a graph and locations of objects can be defined by the location within that graph. The mapping from raw coordinates, which by themselves do not include the location within a graph, to the location within the graph is called map matching [RH10]. Those networks have certain characteristics which distinguish them from unrestricted or less restricted environments. For example, the distance between two points in the graph is not equal to the air-line distance, wherefore proximity calculations differ in such networks [RH10].

Other objects, such as pedestrians, plains and vessels are less restricted. They often have certain paths to follow, nevertheless, they are technically not restricted to a certain network. However, these objects can be restricted to certain areas. For example, vessels cannot leave the water and it can be forbidden for them to enter some restricted areas [VVBB12]. Pedestrians can be restricted by obstacles such as buildings. This influences, for example, the prediction of locations and the calculation of distances. This work focuses on moving objects that are not strictly restricted by a network. The main targeted moving objects are vessels, without limiting the concepts to this type of vehicle.

2.2 Data Stream Management Systems

Data streams have specific challenges and offer new opportunities for data management and query processing. Querying live data streams in near real-time offers results while things are happening. Managing data streams and allowing flexible and efficient query processing is the topic of DSMS research. DSMS differ from traditional DBMS. The difference can be seen by comparing Figure 2.1 and Figure 2.2.

A DBMS, depicted in Figure 2.1, runs one-time queries on a database. When a query is posed, the query runs for a short time and the result is calculated. Eventually, the result is provided and the query is removed from the system. The database is changed by insert, update and delete statements, but for a single query, the database seems to be not changing as long as the query is running.

A DSMS, depicted in Figure 2.2, in contrast runs continuous queries on continuous data streams. Active data sources, such as moving objects, push their data to the DSMS. The user poses continuous queries, i. e., long running queries that exist in the system and produce results when new data arrives in the system. They run until they are explicitly

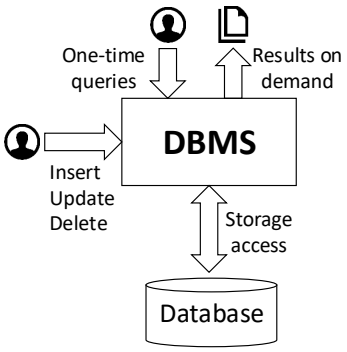


Figure 2.1: Schema of a DBMS. Figure based on [KS09].

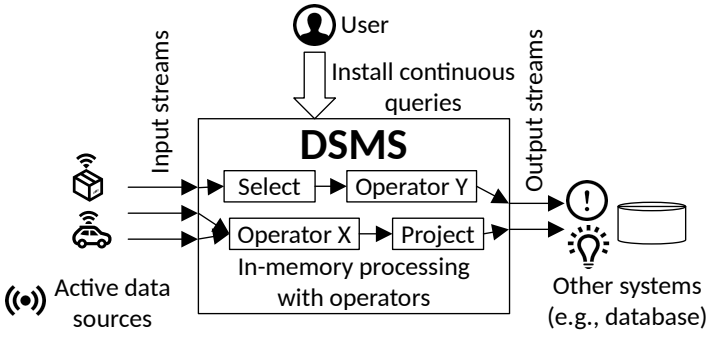


Figure 2.2: Schema of a DSMS. Figure based on [KS09].

removed. This push-based processing mechanism is also called “event-driven” [Luc02]. Data stream elements are denoted as events, i. e., data that is pushed into the system is made of events and the results that are produced by the system are also events. When a new event enters the system, the continuous query instantly processes the new data and produces new results. There does not need to be a timer for which the query waits, for example, if the results would be produced every five minutes or so (even though this would also be possible).

The processing of the queries typically run in-memory. The data flows through the operators and are eventually send to other systems that use these results. This could be a user interface for a user or a database that stores the results.

2.2.1 Operators and Operator Graphs

To process queries on data streams, DSMS represent queries with operator graphs. An example operator graph is depicted in Figure 2.3. Such an operator graph consists of operators as nodes and connections between these operators as vertices. The graph is typically acyclic and directed [CHKS04], however, cyclic graphs can be possible [Bol11]. Operators subscribe to the output of other operators and subsequently receive the others operators results. Hence, data stream elements flow from one operator to the next. The operators work on the data they receive and send their results to the subsequent operators in the graph.

At the beginning of a query graph are one or more sources where the data streams from other systems are received. In the example in Figure 2.3, the data stream elements from Source 1 are projected, i. e., the schema of the tuples is changed. The window operators for Source 1 and Source 2 limit the view of the streams before the join operator joins the two streams. The last operator uses a select predicate to filter out the tuples that do not fulfill the predicate. At the end is a sink where the results leave the system, for example to be written to other systems.

Typical operators that are used in the context of data streams are also known from relational DBMS. Some of the most common operators are the selection-, join-, projection- and aggregation-operator [CHKS04]. In the context of data streams, in contrast to databases, window operators are important as well. They allow to define windows as described in Section 2.2.2. A DSMS could also offer other operators with more application specific logic, e. g., for machine learning [GHN14].

Even though the operators are connected to each other in the query graph, a single operator has no knowledge about the whole plan, i. e., an operator does not know what (kinds of) other operators are in the plan. Still, the plan can be optimized during the translation from the algebraic definition to the physically executed plan due to the algebra used for the definition. With an algebraic optimization, the order of the operators can be changed to improve the efficiency without changing the output of the plan.

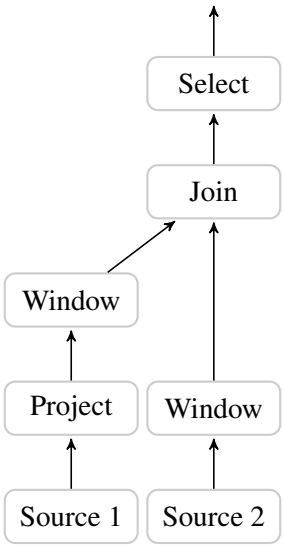


Figure 2.3: Example of an operator graph with the data flowing from bottom to top.

2.2.2 Windows

When querying data streams, often only the current data is of interest. For example, when aggregating a value such as the average speed of a moving object, the data elements from years ago are irrelevant if the current situation needs to be observed. Windows are a concept to limit a data stream to a certain part of it. In this case, it could be used to calculate the average speed of a moving object during the last ten minutes. Additionally, this concept helps to avoid memory leaks. In a potentially unbounded stream it is infeasible to store all historic data and it is not possible to wait for all data to arrive before starting to calculate a queries' result.

The concept of windows is depicted in Figure 2.4. New data stream elements flow into the system from the right. When they arrive at the window, the window is updated: old elements may be removed and the new element is added. The results of the subsequent operators are updated accordingly.

The definition of a window can be formulated in various ways, based on the need of the data processing. One possibility is to hold a certain time range, e. g., the newest ten minutes. Such a time window can hold a variable number of tuples, as the data rate can change over time. In contrast, a window can be defined to hold a predefined number of tuples, e. g., the 100 newest ones. Another possibility is to define a predicate which needs to be true for the tuples within the window. It could be defined that the sum of a specific value in the tuples needs to be lower than 100. If a tuple with a high value arrives in the window, many old tuples could be dropped.

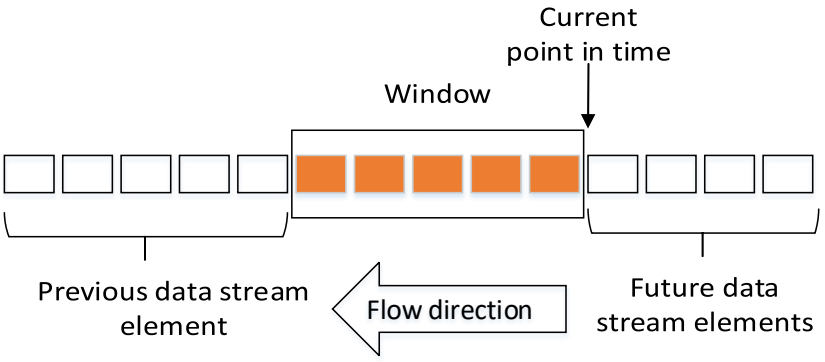


Figure 2.4: Window on a data stream. Figure based on [Bol11].

The advance of a window can be defined in various ways. In general, the window has a certain size s and a certain window advance or slide granularity a. For example, the size of a window could be 100 tuples and the advance 1 tuple, hence $a = 1$. With this setting, the window is *sliding*: it advances with the smallest possible steps [KS09]. This can, for example, be used on a stream that contains the speed of a moving object. When an aggregation calculates the average speed in the last hour and a new output is necessary at every new stream element, a sliding window is applied before the aggregation operator.

If $1 < a < s$, i. e., the window advances only every few tuples but more often than the size of the window, the window is *jumping*. Some possible window states are left out, but every tuple is at least in one window. Following the previous example, the average speed of the object in the last hour needs to be known, but an update is only desired every ten minutes.

If $a = s$, the window is *tumbling* [KS09]. With this setting, every tuple is in a window exactly once and the windows are not overlapping. Here, the average speed for the last hour would be calculated exactly once per hour. If $a > s$, some tuples are not in a window at all, hence, there are gaps between the windows. To follow the previous examples, in this case, the average speed would be calculated, for example, every two hours for the last hour.

There are multiple known and tested possibilities to achieve the explained window behavior. Common window approaches are the positive-negative approach [HAF+03] and the interval approach [KS09, Kuk15]. With the positive-negative approach, for each data stream element two messages are send through the query graph: at first, the stream element is send through the query graph with a positive sign to state that the data stream element is now valid. Later, when the data stream element is invalidated, a message with a negative sign (and the same data stream element identifier to match the messages) is

send through the query graph. The messages are send by a window operator, which is placed before the operators that need the window logic in the operator graph.

The other popular approach is the interval approach. Here, a window operator attaches an end-timestamp to each data stream element so that every stream element has two timestamps: a start timestamp that states when the element starts being valid and an end-timestamp that states when the element stops being valid. The time progress is defined by the newly incoming data stream elements. The operators in the query graph use these timestamps to decide which elements to keep and which to discard.

Other approaches define batches for every window so that data stream elements are collected in batches. Note that for a sliding window, a data stream element can be in multiple batches. These batches can then be processed, for example they can be aggregated. An example for a system with this approach is Apache Flink[2] [CKE+15a]. Another possibility is to not have a special window operator but to have the window logic within the operations that work with the data [BBMS05, AAB+05, BBF+10].

2.3 Formal Definition of Data Stream Processing

The characteristics of moving object data streams are described in Section 2.1. In this section, the formal definition of the data stream is given and the implications for the processing are explained. Data stream operators are defined on logical data streams, i. e., describe the set-theoretic function. The physical operator later describes how the operator is implemented with an algorithmic view over physical data streams. This approach has some advantages: the formal definition of the operations on the data stream allow a deterministic processing of the defined queries and with that, repeatable results. An important characteristic for this approach is the snapshot-reducability.

2.3.1 Snapshot-Reducability

A snapshot of a logical data stream is the set of tuples that are valid at a certain time instant. A data stream can be sliced into a series of snapshots. A data stream operator is snapshot-reducable *iff* for every snapshot of a data stream the results of the operator are equal to the non-streaming relational counterpart of the operator [KS09].

This is an important property of the data stream processing in DSMS that rely on a formal relational algebra. It allows to define stream operators on non-temporal sets of tuples and use these operators in a temporal data stream. Another conclusion is that the results of operators are deterministic: on the same set of valid tuples all operators need to produce the same output, regardless of the previous input.

[2] https://ci.apache.org/projects/flink/flink-docs-master/dev/stream/operators/windows.html

2.3.2 Logical Stream

To achieve snapshot-reducability and use the characteristics of this approach, this thesis uses the same definition for logical and physical data streams as proposed by [KS09]. To use the time-interval approach, the time domain needs to be defined. "Let $\mathbb{T} = (T, \leq)$ be a discrete time domain with a total order $\leq [\dots]$. A *time instant* is any value from T." [KS09]. The data stream consists of data stream elements. A data stream element is of type $\mathcal{T}$. $\Omega_{\mathcal{T}}$ is the "set of all possible tuples of type $\mathcal{T}$" [KS09]. For example, the type $\mathcal{T}$ could be a relational tuple. In this case, the tuple has a certain schema. As the logical stream can possibly have multiple equal elements but is a mathematical set, the number of equal tuples need to be denoted as well (the multiplicity). This is done with n.

Definition 2.1. (Logical Stream [KS09]): A logical stream S^l of type $\mathcal{T}$ is a potentially infinite multiset (bag) of elements (e, t, n), where $e \in \Omega_{\mathcal{T}}$ is a *tuple* of type $\mathcal{T}, t \in T$ is the associated *timestamp*, and $n \in \mathbb{N}, n > 0$, denoted the *multiplicity* of the tuple. Let $\mathbb{S}^l_{\mathcal{T}}$ be the set of all logical streams of type $\mathcal{T}$.

2.3.3 Physical Stream

For the physical processing, a more efficient representation of streams is needed. In case that a stream element is valid for multiple points in time, the logical representation would contain a data stream element for each point in time. The physical representation "coalesces identical tuples with consecutive timestamps into a single tuple with a time interval" [KS09]. This is done with the start- and end timestamps t_S and t_E.

Definition 2.2. (Physical Stream [KS09]): A physical stream S^p of type $\mathcal{T}$ is a potentially infinite, ordered multiset of elements $(e, [t_S, t_E))$, where $e \in \Omega_{\mathcal{T}}$ is a tuple of type $\mathcal{T}$, and $[t_S, t_E)$ is a half-open time interval with $t_S, t_E \in T$. A physical stream is non-decreasingly ordered by start timestamps.

2.3.4 Transformation

The operations on data streams are defined on logical streams while the physical streams are the basis for implementation. To show that the physical representation is semantically equal to the logical definition, a transformation from physical to logical streams needs to be given. Here, the transformation by [KS09] can be used:

Definition 2.3. (Physical Stream To Logical Stream [KS09]): $\varphi^{p \mapsto l}(S^p) := \{(e, t, n) \in \Omega_{\mathcal{T}} \times T \times \mathbb{N} \mid n = |\{(e, [t_S, t_E)) \in S^p \mid t \in [t_S, t_E)\}| \wedge n > 0\}$

The time interval in the physical stream is split into single logical stream elements. The counter n is derived by counting all physical stream elements with the same tuple e at the same logical timestamp t.

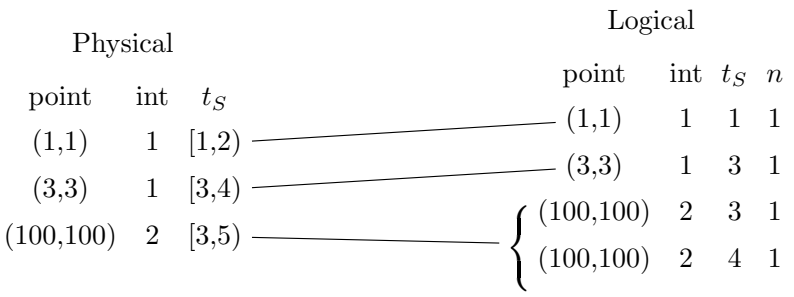


Figure 2.5: Example transformation from a physical to a logical stream

Figure 2.5 depicts an example transformation: a physical stream with three stream elements is transformed to a logical stream. The first two physical stream elements are only valid for one chronon and therefore result in only one stream element in the logical view. The last physical stream element is valid for two time instances. This stream element results in two logical stream elements, one for each point in time. n is always 1 because there are no two or more equal stream elements which need to be distinguished with this counter.

2.3.5 Chronon Stream

In a chronon stream, every data stream element is only valid the minimal possible time interval, i. e., t_S and t_E are two consecutive points in time with the granularity of the used time units [KS09].

In this thesis, typically every moving object measures its own location once in a while and reports this location to the DSMS immediately. As the objects are continuously moving, the location measurement is only valid the minimal possible time interval or, in other words, one time instant. Hence, it is a chronon stream.

2.3.6 Trajectory

In the context of moving objects, the term trajectory is commonly used to describe the path of a moving object over time [FXX+13, PAA+17]. A stream of trajectory data typically consists of stream elements with a spatial object (e. g., a point or a region), an identifier for the moving object and the timestamp of the measurement that states when the object was at that location. As the locations are only known at certain points in time, the locations in-between are often simply connected by a straight line [Gal16].

Figure 2.6 shows an example of a trajectory. The moving object, here a point, moves through the space over time. Each measurement is depicted with a dot and its respective

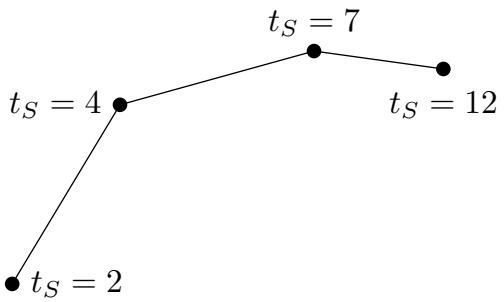


Figure 2.6: Example of a trajectory

timestamp. The known locations are connected with a straight line, as the exact locations at these times in-between the measurements are not known.

2.4 Spatial Data Processing

This section gives an overview of some basic geospatial foundations used in Geographic Information Systems (GIS). It describes, how spatial information can be represented and what operations can be calculated on this data. Additionally, a short introduction to spatial indexing methods is given.

2.4.1 Spatial Data

Data created by moving objects contains spatial information. That can be the location of the object, the direction of movement or the extend of the object. The reference system of the moving object could be local, e. g., a football field [GFW+11], or global, i. e., the earth. Data that is referenced to the earth can be called geospatial data [LGMR10] and is the main focus of this work.

Today, GNSS such as GPS is often used to obtain location information, for example, from navigation systems in vehicles. Typically, this data is given in latitude and longitude values using WGS 84 as the datum [EPS07]. WGS 84 defines the reference ellipsoid, hence, the exact axis lengths of the earth and the flattening of the oblate ellipsoid. The datum is used to map from the coordinate system to the earth. Together with the coordinate system, these build a coordinate reference system. Throughout this work, WGS 84 is used at the geodetic datum, as AIS locations are measured using GPS.

The flattening is used because the earth is not a perfect sphere, but is instead a little flattened due to its rotation. “The Earth is slightly flattened, such that the distance between the Poles is about 1 part in 300 less than the diameter at the Equator.” [LGMR10]. A flattened ellipsoid is depicted in Figure 2.7. The ellipsoid is rotating around the minor axis. The distance between the poles is smaller than the diameter of the ellipsoid at the

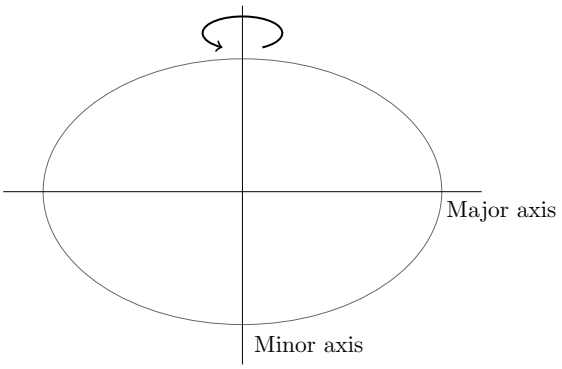


Figure 2.7: Flattened ellipsoid, similar to [LGMR10]

major axis (i. e., at the equator). Nevertheless, the flattening of the earth is way smaller than depicted here.

With latitude and longitude values, single points on the surface of the earth can be referenced. Nevertheless, spatial objects can also have an extend. A common standard for spatial objects is the Open Geospatial Consortium (OGC) Simple Feature Access (SFA) [IH11]. The description here refer to version 1.2.1 of the SFA. The class hierarchy of the `Geometry` class, which is the abstract parent class of all geometries in the SFA, is depicted in Figure 2.8. It defines a geometry object model with points, curves and surfaces. All objects are created using points. For example, a `LineString` contains multiple `Point` objects. The SFA does not know rounded objects, i. e., all points are always connected using straight lines. With this object model, two-and-a-half dimensional (2.5D) objects can be represented. This means that for each (x, y) coordinate, only one z coordinate can exist [LGMR10], i. e., each point has exactly one elevation value. Phenomena such as bridges, where at the same (x, y) coordinate two elevations exist, cannot be represented with a 2.5D model.

2.4.2 Topological Predicates

Topological relationship predicates between spatial objects in the SFA can be expressed with the Dimensionally Extended Nine-Intersection Model (DE-9IM), which defines nine intersections of two spatial objects. The intersections are defined for the interior, boundary and exterior of the spatial objects, resulting in nine combinations of intersections. Figure 2.9 gives two examples. In Figure 2.9a, the interior of a polygon is intersected with the interior of another polygon, resulting in a new polygon (orange dashed area) as the result. In Figure 2.9b, the interior of the left polygon is intersected with the boundary of the right polygon, resulting in a line, i. e., a one-dimensional result, in contrast to the two-dimensional result of the intersection operation in Figure 2.9a.

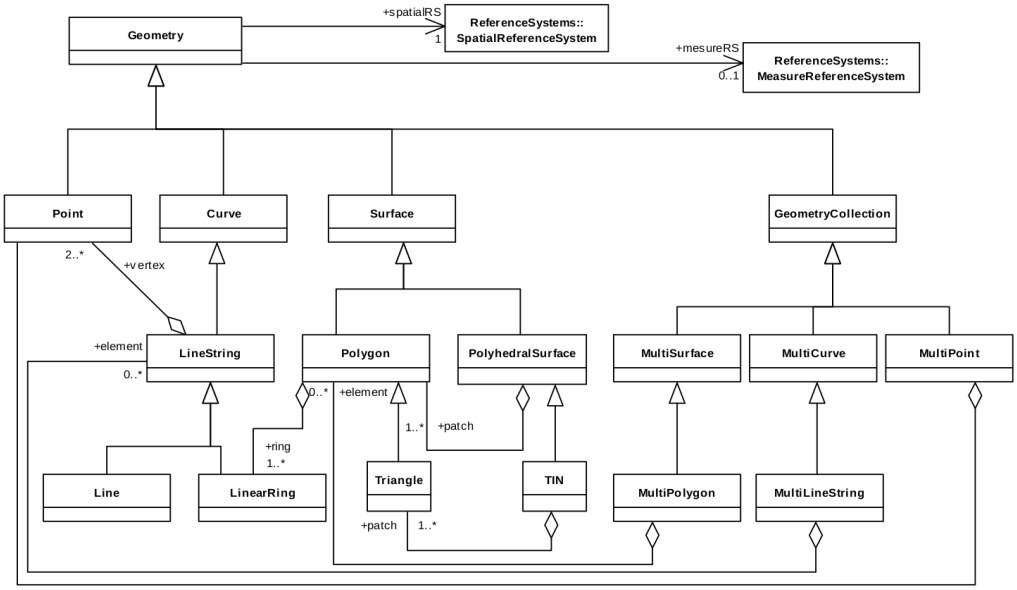
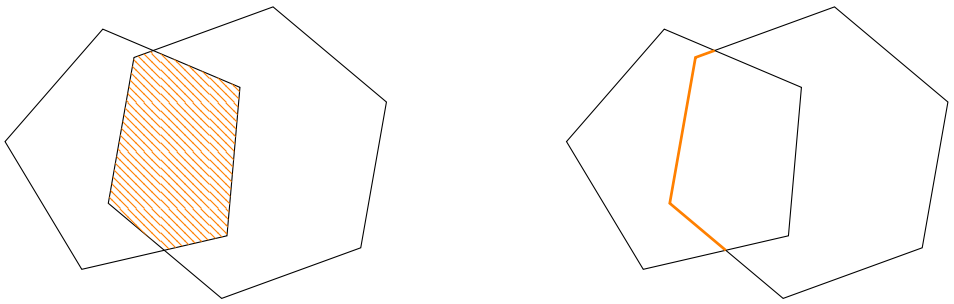


Figure 2.8: Geometry class hierarchy of the OGC SFA object model in version 1.2.1. Figure by [IH11].



(a) Intersection between the interiors of the left and right polygons.

(b) Intersection between the interior of the left and the boundary of the right polygon.

Figure 2.9: Example of the DE-9IM

Based on the DE-9IM, "a set of named spatial relationship predicates" [IH11] is defined for simpler use in spatial relationship predicates. For example, with the "within" predicate, it can be tested if one object is within the boundaries of another object. The SFA geometry model is for example implemented in the commonly used JTS Topology Suite (JTS). Next to the geometry description, it also contains the definition of topological predicates.

2.4.3 Geospatial Distance Calculations

The previous operations did not include distance calculations to know the distance between two points on the surface of the earth. As the earth is not a plane, Euclidean distance calculations lead to wrong results, especially for longer distances. Spherical or ellipsoid distance calculations are used instead. They are computationally more expensive, but lead to more accurate results.

Examples for more accurate formulae for distances on the earth are the haversine formula and the Vincenty's formulae. The haversine formula simplifies the earth to a sphere. Hence, it calculates the spherical distance, which can lead to a slight error of a few kilometers, depending on which sphere is used to approximate the earth and the locations of the two points for which the distance is calculated [Hub12].

Computationally more expensive but more accurate are distance calculations which approximate the earth as an ellipsoid with a flattening (mostly of about $1/300$). The Vincenty's formulae can be used to calculate these distances with an error up to about 0.5 mm on the ellipsoid [Vin75]. Hence, when using an ellipsoid that is an accurate representation of the earth, these formulae lead to very accurate results.

2.4.4 Spatial Queries

In spatial database systems, such as PostGIS[3], spatial queries can be defined with SQL as the query language. The supported spatial data types and operations are often based on the OGC SFA[4] (cf. 2.4.2). An example query is given in Listing 2.1. Here, the Maritime Mobile Service Identitys (MMSIs), i. e., the identification numbers of vessels, which are within the borders of the country with the name "Germany", are selected. The spatial operation is the "ST_Contains"[5] operation, which calculates if the geometry (cf. Figure 2.8) of a vessel (e. g., a point geometry) is contained in the geometry of the country (e. g., a polygon geometry).

[3] `https://postgis.net/`

[4] `https://postgis.net/features/`

[5] PostGIS uses the naming convention from SQL/MM and thus starts the geometry functions with an ST prefix (see `https://postgis.net/docs/reference.html`). SQL/MM is an extension to the OGC SFA standard, wherefore the functions are mainly compatible with the SFA standard [Sto03].

```
1 SELECT vessel.mmsi
2 FROM country, vessel
3 WHERE ST_Contains(country.geom, vessel.geom)
4 AND country.name = 'Germany';
```

Listing 2.1: A spatial query in PostGIS

This is a quite simple spatial query. It is obviously not processing spatio-temporal data, i. e., it does not use any temporal data of the relations. Additionally, this query is a traditional one-time query: it is started, processes its results, returns these and is then no longer active. Hence, the results are not updated when new data is added. Nevertheless, when starting such a query, the results have to be processed quickly, even if the amount of data in the database is very high. For this purposes, spatial indexes are used.

2.4.5 Spatial Indexes

To reduce the costs of a query, expensive spatial calculations need to be avoided as far as possible. To achieve this goal, the question arises how to decide which elements are within the query result with as few spatial calculations as possible (because spatial calculations are costly [BKS93]). A common approach is to split the calculation of the results into one or more filter steps and a refinement step [BKS93, Gü94, Bri07]. The goal of a filter step is to reduce the potentially huge number of possible candidates to only those which are in the final result of the query [Bri07]. Doing this, the costs of the filter steps need to stay below the costs of the actual calculations. This is achieved by approximation techniques, mainly with the support of spatial indexes [Bri07].

The resulting candidates which went through the filters (and are thereby potentially in the result set of the query) are then refined in the refinement step. Here, the exact calculation is done on the potentially way smaller set of data. The general approach is depicted in Figure 2.10 from [Bri07]. At the beginning, before any filter step, there is a large set of candidates (c_0). Each filter step reduces the number of candidates ($c_1, c_2, \ldots$). Doing this, some elements can be safely put in the set of result elements ($r_1, r_2, \ldots$) and some can be filtered out ($e_1, e_2, \ldots$). Elements that are filtered out late in the filter process (e. g., in the refinement step) are false hits and are aimed to be as few as possible [BKS93]. The computationally cost of the operations are typically increasing during the process. For example, in a setup with one filter step and one refinement step, the filter process should be (way) cheaper than the refinement step.

Spatial index structures in databases are, for example, space-filling curves [RSV02], grids [RSV02], quadtrees [FB74], and R-trees [Gut84]. One-dimensional index structures such as the B-tree [BM72] perform poorly on multi-dimensional, and with that, spatial data [LGMR10]. A spatial index needs to be build and maintained when the un-

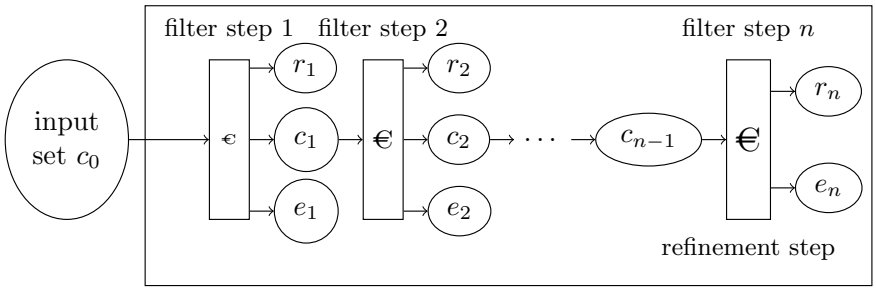


Figure 2.10: Steps in the filter and refine process. Figure reproduced based on [Bri07]

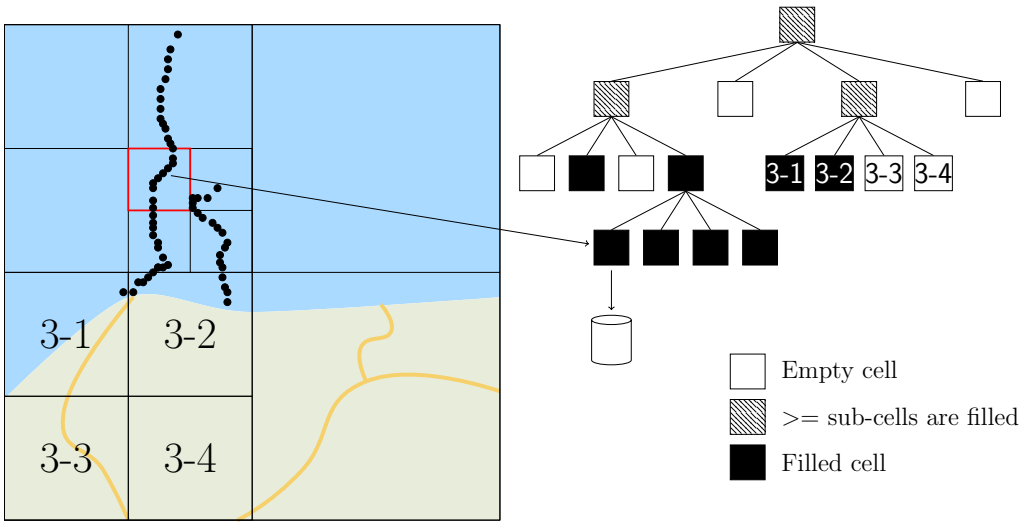


Figure 2.11: A spatial area covered by a quadtree. Figure reproduced based on [BG18]

derlying data changes. In the following, a quadtree is used as an example for a spatial index.

The basic approach of a quadtree is shown in Figure 2.11. On the left, a region (depicted is a coast line and locations as points from moving objects) is covered by a quadtree. This quadtree splits the quadratic regions into four new regions based on a certain rule, for example, if the number of stored spatial objects in a region reaches a certain limit. In this example, this has been the case for the left cells, where all stored points (the black dots) occur. The very dense area (1-4) was split again because of the high number of objects.

On the right, the resulting tree is shown. The leaves point to where the spatial objects in this particular region are stored. Hence, if a query needs to know the objects in a certain region, it is not necessary to go through all stored objects. The numbers in the region part 3 are shown to depict which regions belong to which leaves.

In this simple example, the cells are always split in four equally sized quadratic cells. This does not need to be the case. The nodes in the tree can instead store the split point. From there, the rectangular region is split into four new rectangles. This can lead to a better balance of the data in the index, i. e., less spatial objects being stored in the same leaf in densely populated areas while others stay nearly empty [Oos99, LGMR10].

2.4.6 Spatio-Temporal Queries

Spatio-temporal queries take an additional temporal dimension into account when processing spatial queries. The spatial data has an additional temporal attribute to state the time at which the spatial object is valid. For example, a trajectory from a moving object is made up of a list of points at different, increasing, points in time (as one moving object typically cannot be at different locations at the same point in time). In Listing 2.2, an example of a spatio-temporal query in a database is presented to give the reader an idea on how such a query can be formulated and how the data and results can look like.

For this example, the temporal support of PostGIS is used, more specifically the `ST_ClosestPointOfApproach` function. This function takes two trajectories as the input. A trajectory is a linestring with a start and end time. The values, i. e., the locations in-between the given points of the linestring are linearly interpolated[6]. The linestrings are created from Line 3 till Line 18 and use the WGS 84 reference ellipsoid.

While creating the linestrings, the measurement field `m` is exploited to store the time at which the moving object is at certain locations on its path. The `m` field is a generic field for spatial data objects to store "measurements". Hence, this field can be used for many different use cases, for example, to store temperatures for certain locations. In this case it is used for the timestamp, i. e., the temporal dimension. This is done with the `ST_AddMeasure` function. It adds a value `m` to a linestring. The value in `m`, a timestamp, is interpolated between the start and end value given in the arguments of the function. This way, each point on the linestring has a certain timestamp in the `m` field.

In Line 20 the Closest Point of Approach (CPA) is calculated using the aforementioned function. Lines 22 and 23 get the actual points (`pa` and `pb`) from the original linestrings `a` and `b` with help of the function `ST_LocateAlong` and the previous result `m` (`m` only points to the correct point by the time dimension). Finally, in Line 28, the distance between the two CPA locations is calculated. The result of this query, a tuple with timestamp and distance, is shown in the following table:

t	distance
2018-07-25 11:31:46.158252+02	10797.8010620825

The distance is given in meters. Hence, at about 11:31 o'clock, the two moving objects reach the CPA with a distance of about 10.8 km. Note that this is a result of a database

[6] `https://postgis.net/docs/ST_AddMeasure.html`

on static data, i. e., there are no input data streams nor do the results update in an event-driven manner.

```
-- Calculate the time and distance where two moving objects are closest to each other
WITH inp AS ( SELECT
  ST_AddMeasure(
      -- geometry from west to east (long, lat)
      ST_GeomFromEWKT('SRID=4326;LINESTRING (6.931236 53.775987, 8.787664 53.990979)')::geometry,
      -- measure start
        extract(epoch from '2018-07-25 10:00'::timestamptz),
      -- measure end
        extract(epoch from '2018-07-25 12:30'::timestamptz)
  ) a,
  ST_AddMeasure(
      -- geometry from south to north (long, lat)
      ST_GeomFromEWKT('SRID=4326;LINESTRING (8.166653 53.584027, 7.886626 54.176583)')::geometry,
      -- measure start
        extract(epoch from '2018-07-25 10:00'::timestamptz),
      -- measure end
        extract(epoch from '2018-07-25 14:00'::timestamptz)
  ) b
), cpa AS (
  SELECT ST_ClosestPointOfApproach(a,b) m FROM inp
), points AS (
  SELECT ST_Force2D(ST_GeometryN(ST_LocateAlong(a,m),1)) pa,
         ST_Force2D(ST_GeometryN(ST_LocateAlong(b,m),1)) pb
  FROM inp, cpa
)

SELECT to_timestamp(m) t,
       ST_DistanceSpheroid(pa,pb,'SPHEROID["WGS 84",6378137,298.257223563]') distance
FROM points, cpa;
```

Listing 2.2: A spatio-temporal closest point of approach query in PostGIS based on the example query in the PostGIS documentation: `https://postgis.net/docs/ST_ClosestPointOfApproach.html`

2.5 Moving Object Algebra

Queries on moving objects have been researched before, mostly for DBMS [GAA+05] but also for DSMS [GMKB12]. An important foundation for querying moving objects is the moving object algebra by Güting et al. [GBE+00] on which other works are build [GMKB12].

a *point* value a *points* value a *line* value a *region* value

Figure 2.12: The spatial data types [GBE+00]

The moving object algebra uses the concept of the second-order signature by [Gü93]. Every signature consists of sorts and operations. The first signature defines the type system of the algebra with collections of types (sorts) and the constructors (operations). "The second signature defines the operations over the types of the first signature." [GAA+05].

The first signature defines a number of spatial and non-spatial types. The spatial types are two-dimensional points (both singular and plural, i. e., *point* and *points*), *lines* and *regions* (depicted in Figure 2.12), but standard data types such as *real*, *integer* and *boolean* are also part of the type system. Note that a line as well as a region can be seen as an infinite amount of points, i. e., a *point set*.

Based on these types with their respective constructors, non-temporal unary and binary operations can be defined. Examples for binary operations are the topological *in_interior* and the metric *distance* operations [AGB06]:

$point \times region \rightarrow bool$ **[in_interior]**
$point \times point \rightarrow real$ **[distance]**

The notation shows the second-order signature of the operations. It describes two input types and the output type of the operation. The name of the operation is on the right. The algebra defines the behavior of the non-temporal operations for a number of different classes of operations, e. g. the topological *in_interior*, the numeric *area* (which returns the size of a region), the *distance* (e. g., between two points) and *direction* operation [GBE+00].

Up until now, the types and operations are only static, i. e., non-temporal. Nevertheless, points, lines and regions can move and typically do this in a continuous manner [GBE+00]. For this purpose, the algebra contains temporal types. These are derived from the non-temporal types with the *moving* type constructor. This constructor takes a type and creates a temporal version of it. For example, the *point* type becomes an *mpoint* type. The *mpoint* is basically a function that is valid within a defined period of time. For a given time instant (which is also a type), it returns a *point*. This is true for other types as well: there exist temporal versions of regions, integers, booleans, etc. [GBE+00].

The operations of the algebra are defined on the non-temporal types. This is the so-called kernel algebra. With an approach called *lifting*, the non-temporal operations are lifted to the temporal counterparts. Basically the operations are lifted so that if any

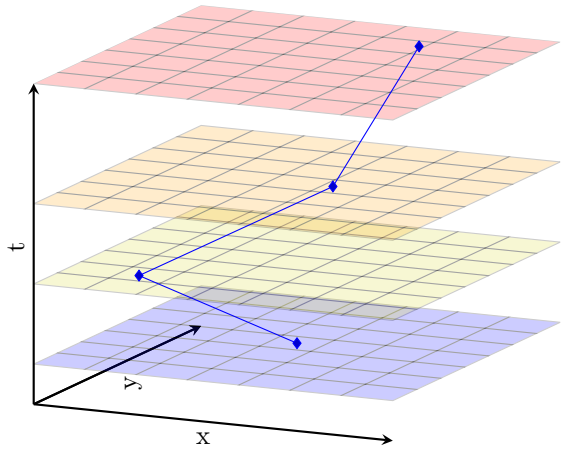


Figure 2.13: Sliced data from a moving point in time, similar to [AGB06]

argument of an operation is temporal, the result of that operation will be temporal, too. Any combination of temporal and non-temporal arguments is possible [GBE+00]. For the example above, the lifted versions of the operations are as follows [AGB06]:

$mpoint \times region \rightarrow mbool$	**[in_interior]**
$point \times mregion \rightarrow mbool$	**[in_interior]**
$mpoint \times mregion \rightarrow mbool$	**[in_interior]**
$mpoint \times point \rightarrow mreal$	**[distance]**
$point \times mpoint \rightarrow mreal$	**[distance]**
$mpoint \times mpoint \rightarrow mreal$	**[distance]**

Considering the first example, the *in_interior* operation is used on a moving point and a non-moving region. A point can be inside or outside of a region and when moving, this state can change. The point can move from inside the region to outside the region, wherefore the result of the operation will change over time. Therefore, the result of that operation is not a *bool*, but an *mbool*, i. e., the temporal version of the *bool*.

As described in Section 2.3.5, the location of a moving object changes continuously but is only known at certain points in time to the DSMS or DBMS. In [AGB06], this is called a *sliced* representation. The location in-between the slices needs to be calculated. In the moving object algebra, this is done by the temporal function. This function can calculate the location of the objects (e. g., an *mpoint*) in-between the slices. For example, a linear function can be used [AGB06]. Figure 2.13 depicts the sliced representation. There are four slices over time (which are not necessarily in the same distance from each other). In every slice, the location of the moving point is marked with a dot in the two-dimensional space (x and y). The location of the moving point in-between the slices can be calculated with the temporal function. In this case, it is a linear function to connect the sliced points with a straight line.

As seen above, when querying with this algebra, temporal types can be the result. For example, when querying "From when to when was the airplane X (mpoint) above Germany (region)?", the result of the *in_interior* would be an *mbool*. Such a temporal type consists of temporal functions, or, as another logical representation, of "infinite sets of pairs (instant, value)" [GBE+00]. The time in which the function is valid is limited: there is a starting and an end point of the validity of the function. To answer the query above, these temporal points are needed. They can be derived with the *initial* and *final* operations which take a temporal type and return an (instant, value)-pair, from which the time (instant) can be derived. This approach is similar to the time-interval approach by [KS09] where each tuple contains the time interval in which it is valid.

The *atinstant* operation uses a temporal type, e. g., an *mpoint*, and an instant of time to get the non-temporal type at that instant of time. With that operation, the temporal function is used to calculate the value of the temporal type at the given instant of time. These instants can be the areas in-between the slices in Figure 2.13.

The mapping of the types and operations to a DSMS needs to consider some of the characteristics that differ between DSMS and DBMS, which is an important part of the concept in the next chapter. To give a better understanding of moving objects in general, a real world use case is introduced in the next section.

2.6 Location Based Services and Vessel Traffic Services

Location Based Services (LBSs) is a general term for services that use the location of the services' user, e. g., through the GNSS sensor of a mobile phone. Finding nearby points of interest, nearby friends or playing location-aware mobile games are typical examples for LBSs. Additionally, more critical services such as air traffic control, locating a phone call in an emergency situation or finding stolen cars also fall under the general term of LBSs [SV04].

To implement these services, spatio-temporal queries have to be executed, sometimes with data from multiple moving objects. The queries have to update their results when the user or the queried objects move and the results should be up-to-date to be aware of the current context of the user. Hence, LBSs are in general a category of application where spatio-temporal data streams from moving objects have to be processed. Vessel Traffic Services are a part of LBSs in the maritime domain.

2.6.1 Vessel Traffic Services

The research described in this thesis aims to be generic and applicable for moving objects in general. Nevertheless, to have a more specific application example, the traffic at sea is the use case referred to in this work. Next to the vessels themselves, cost guards, shipping companies and Vessel Traffic Services (VTSs) are working with data about

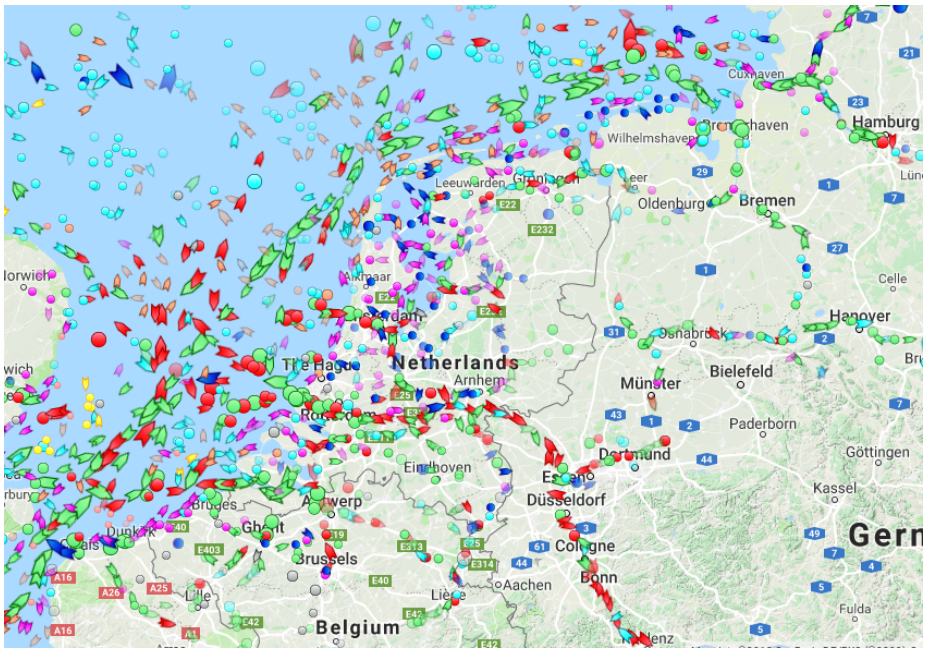


Figure 2.14: Traffic at the North Sea and the English Channel. Screenshot from `https://marinetraffic.com/` with map background data by Google.

the locations of vessels [SM07, EEB+07]. VTSs are onshore systems which manage traffic in extensively used or otherwise difficult waterways. They provide information to vessels, such as warnings about harsh weather conditions and positions of other vessels.

To ensure a safe and efficient traffic flow, VTSs rely on multiple data sources and communication infrastructure with the vessels. Typical data sources are radar sensors and AIS [LH06]. The data is used to keep an overview of the current situation at sea, avoid collisions and search for missing vessels. Depending on the area, the number of vessels can be very high. For example, the traffic on the North Sea and especially in the English Channel is very dense, as can be seen on the screenshot in Figure 2.14.

The high traffic density and with that the huge amount of data in this example let the question arise how automatic data processing can be applied on this data, especially on AIS data. Automatic processing can help to get a better overview of this data to increase the situation awareness, what is especially useful for VTS applications. Additionally, storing and analyzing the data can help to make decisions about port development and waterway policies [Tso16].

For a VTS, the current situation at the waterways is very important. This has to be considered for solutions that automatically process the data from the vessels and potentially other data sources, such as weather sensors. This live processing of the moving object data is the motivation of this work.

2.7 Related Work

Moving object data is a popular spatio-temporal data type and solutions for different challenges with this specific kind of data have been researched. Historically, the moving object data was available in databases as static trajectories, i. e., sets of historic moving object data. SECONDO [AGB06] is a popular example for a moving object database. It allows to do operations on the data that are specific for moving objects. For example, with the system it is possible to query all occurrences when trains (i. e., moving objects) passed a certain area [AGB06]. The spatio-temporal operations are integrated through algebras. In this case, a spatial and a temporal algebra are integrated into the base system. In contrast to this work, SECONDO does not aim at data streams. The queries performed on the system are one-time queries and do not run continuously.

Newer approaches allow to run continuous queries on data streams from moving objects. They develop their own streaming system or use an existing streaming system and extend it by spatio-temporal functionality.

Continuous queries on data streams from moving objects can be processed with the open-source project "Tile38"[7]. It is designed as a database with a spatial index but supports some kinds of continuous queries to get alerted when a moving object is within a certain area or close to another moving object. In contrast to this work, "Tile38" does not build on a DSMS with a clear streaming approach and an operator-based querying language. Hence, it is not as flexible, for example, to fuse different data streams from different sources.

Galić [Gal16] describes the two streaming systems OCEANUS and MobyDick[8] for moving object data streams. OCEANUS [GMKB12] uses "TelegraphCQ" [CCD+03] as the underlying DSMS and extends it with PostGIS[9] for spatial operations. They use an extension of SQL for the definition of continuous queries. MobyDick extends Apache Flink[10] for distributed stream processing of moving object data streams. MobyDick adds, for example, a data type for temporal points, i. e., a point on a trajectory of a moving object to Apache Flink. Both works differ from this thesis. They do not describe queries between multiple moving objects, e. g., objects that are close to each other. Instead, they can report all objects within a static region (e. g., a certain city district [GMKB12]), close to a point of interest or that travelled a certain distance within a period of time. The proximity queries (close to a point and within a certain area) differ from queries between moving objects in a way that one part of the query is static, i. e., the area does not move and the exact location of it is always known. Additionally, both works do not discuss performance improvements with spatio-temporal filter and refine algorithms. Despite these differences, they have similarities. For example, MobyDick

[7] `http://tile38.com/` and `https://github.com/tidwall/tile38`
[8] `https://bitbucket.org/DarioOsm/mobydick`
[9] `http://postgis.net/`
[10] `https://flink.apache.org/`

has the functionality to predict a moving objects location at a certain point in time (with a linear inter- or extrapolation algorithm[11]).

Other works concentrate on parts of the processing of continuous queries on moving object data streams. [MA08] present a framework for range- and *k*-nearest neighbors (kNN) queries within windows on data streams. They focus on an efficient algorithm that calculates the incremental results. Some temporal considerations that are taken into account in this thesis as well as the focus on the integration into the architecture of a DSMS are not in the scope of [MA08]. [LCY07] also present an efficient algorithm for continuous queries (e. g., a range query) on moving object data streams. They estimate which data is potentially part of future query results and only keep that data in memory while discarding the other data. To do this, future locations of moving objects are predicted with linear extrapolation. Old data is not kept, which is a difference to a DSMS where windows are supported.

Eom et al. [ESL15] propose a query language based on GeoSPARQL for semantic spatio-temporal data streams from Internet of Things (IoT) devices. They use a streaming approach with windows and a special data structure for the spatial data. They combine in-memory storage and database storage for large windows. The system is general purpose and arbitrary queries can be formulated with the given query language. Nevertheless, no queries are described where the locations of multiple moving are compared to each other (e. g., the distance from a moving object to other moving objects). Additionally, it is not described how they would deal with unknown locations for certain points in time. Thirdly, the temporal semantics is given, but seems to be more limited in contrast to this work: the temporal information of a calculated result is not used in the following operations. In contrast, these topics are discussed in this thesis.

Bakli et al. [BSS18] use Hadoop to process data from moving objects. They also use the moving object algebra and integrate support for it into Hadoop. Using Hadoop, the motivation is to process huge amounts of moving object data, similar to this work. Nevertheless, the target scenarios are different. While Hadoop is designed for bulk processing, the focus of this work are near real-time results on live data streams.

[BG18][12] gives an overview of the topic of streaming spatio-temporal data, so-called GeoStreams. Different applications, approaches and techniques are described and an overview of future research challenges in this emerging field is given.

2.8 Summary

This chapter provides background knowledge about the main topics which lay the foundations of this thesis. The characteristics of the data streams that are created by moving

[11] `https://bitbucket.org/DarioOsm/mobydick/src/1b1ceb743ce02206c29679f1265d8018126d39a1/src/main/scala/mobydick/PointUnit.scala`

[12] The author of this thesis also authored this paper.

objects have been explained, as they shape the requirements for the capabilities of DSMS that query these data streams. This thesis aims to extend DSMS, therefore, the basic functionality and technical backgrounds of DSMS have been described here. The operator based, in-memory stream processing and the window logic are the main takeaways from that part (cf. Section 2.2). Based on the general understanding of data streams and stream processing, formal definitions of data streams and important concepts of the processing, such as the snapshot-reducability, are given in Section 2.3.

Spatial data processing is another crucial foundation for this work and has been covered in Section 2.4 with aspects such as spatial data description, spatial operations and spatio-temporal queries. Combining spatial and temporal processing, the moving object algebra is introduced in Section 2.5. This algebra lays an important foundation for this work. It is used for the spatio-temporal operations on moving object data streams. The concept of the sliced representation and the mapping into the streaming environment are explained.

This section is followed by a short introduction about Vessel Traffic Services as a possible application field for this work and to give the reader more practical examples in the remainder of this thesis. Finally, related work is discussed in this chapter in Section 2.7 with SECONDO as a promising system in the field on moving object databases and other works such as OCEANUS in the field of data stream processing and moving objects.

3 Moving Object Stream Query Processing

As motivated in Chapter 1, the goal of this work is to query moving object data streams in a DSMS. The moving object algebra is well-known and precisely defined for the purpose of querying moving objects in databases. It has been used in special-purpose DBMS such as SECONDO [AGB06] as the theoretical foundation. There are also works that use the moving object algebra for spatio-temporal data streams, for example, OCEANUS [GMKB12] (cf. 2.7).

To query moving object data streams, this work combines the time-interval approach by Krämer et al. [KS09] with the moving object algebra by Güting et al. [GBE+00]. The interval approach allows, with its concept of snapshot-reducability [JDB+98], the reduction of streams to snapshot relations, which can be queried using well-known relational algebra. This approach fits well to the approach of the moving object algebra which also uses time-intervals to represent valid-times [JDB+98] of results. Therefore, the interval approach is used in this thesis.

In other words, the connective element of the two approaches is that the concept of snapshots in the time-interval approach can be mapped to the concept of sliced representation of moving objects. The location of a moving object is only known at discrete points in time while in reality, it is moving continuously. The location in-between the measurements is unknown. In the moving object algebra, this is called a *sliced representation*, in the time-interval approach every measurement would be a snapshot. To query the moving objects, it is necessary to know or at least to approximate the locations in-between the measurements. The moving object algebra solves this with a *temporal function*. Adding this capability to a DSMS is an important part of this thesis, as there are important differences between a solution within a DBMS and a DSMS.

How the two theories can be connected is explained in this chapter. Due to the characteristics of data streams, the definition of logical and physical streams needs to be extended to work with moving objects. The so-called bitemporal data stream is defined in Section 3.1. Sections 3.2 and 3.3 go deeper into the moving object algebra within DSMS and explain the type system and the operations on these types for the usage on moving object streams. Parts of this chapter have been described in [BG19].

3.1 Bitemporal Data Streams

Queries in a streaming scenario and in a database scenario have fundamental differences. In a streaming scenario, the data is not known in advance. The stream changes over time while the query is static. In a DBMS, the data is static (at least during the processing of a query). When running a query, it returns a result that is valid at the point in time when the query was executed. To allow a semantically correct and reliable calculation of

results, [KS09] introduced the snapshot-reducability (cf. 2.3.1) which is connected to the requirement that data streams are temporally ordered by their start timestamp (cf. 2.3.3).

The temporal types of the moving object algebra (cf. 2.5) introduce temporal functions which calculate the values of types in-between slices. As the data is not known in advance, in a DSMS it can also be relevant to calculate the location of a moving object in the future, not only between slices. In contrast to the moving object algebra, the end-timestamp of a temporal function may not be defined. Additionally, new data arrives during the processing of the query, which leads to another time domain: real data from the moving objects and predicted data from the temporal functions, while in a DBMS with the moving object algebra, both times can be handled together as one temporal dimension (as the arrival time of the data is not relevant because the data is static).

To handle this difference, the definition of the moving object stream needs to be extended. The possibility to include predicted versions of tuples requires the integration of a new timestamp to state the prediction time. The extended definition of the logical and physical data streams can be done following the approach from Bolles [Bol11]. [Bol11] adds new timestamps to the streams to stay compatible with the stream order required for data stream processing. In contrast to the moving object algebra which uses temporal types, [Bol11] uses a prediction function similar (but not equal) to the temporal function to calculate the values in-between the slices or, in the words of the time-interval approach from [KS09], in-between the snapshots.

To illustrate the problem, let us consider the following situation. A continuous query on a moving object data stream contains the temporal type *mpoint* for a moving object. A query wants to know the distance from that point to a *point* at a certain time in the future, e. g., five minutes from the last slice.

$$mpoint \times point \rightarrow mreal \quad \textbf{[distance]}$$

The distance operation has a temporal real as its result. With an *atinstant* operation on that *mreal* at the temporal instance five minutes into the future, the result would have a future timestamp, ahead of the current stream. That by itself is not a problem, since the stream would still be ordered. The problem occurs when now a new stream element, a real element, arrives that is only one minute ahead of the last real measurement but four minutes behind the predicted element. Now, the temporal order of the stream is disrupted, which is not allowed by the definition of physical streams (cf. Definition 2.2).

This is depicted in Figure 3.1. Based on the tuple at time 4, the value is predicted to time 9. Nevertheless, the next arriving (non-predicted) tuple arrives at time 6. The order of the stream is disrupted, because after a tuple with timestamp $t = 9$ a tuple with a timestamp $t < 9$ arrives.

To prevent this unwanted effect with predicted tuples, [Bol11] introduces the concept of a bitemporal stream with a second timestamp for each tuple. The order of the stream is based on the stream-timestamp t_S while predictions only change the prediction-time

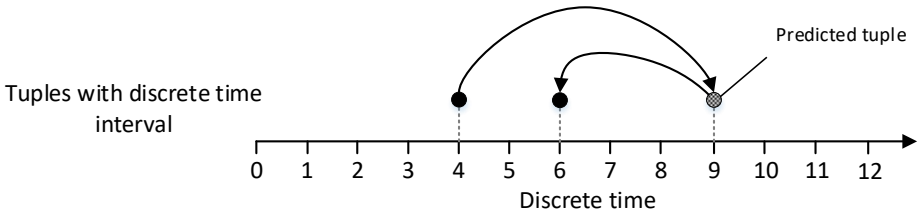


Figure 3.1: Unordered stream due to prediction

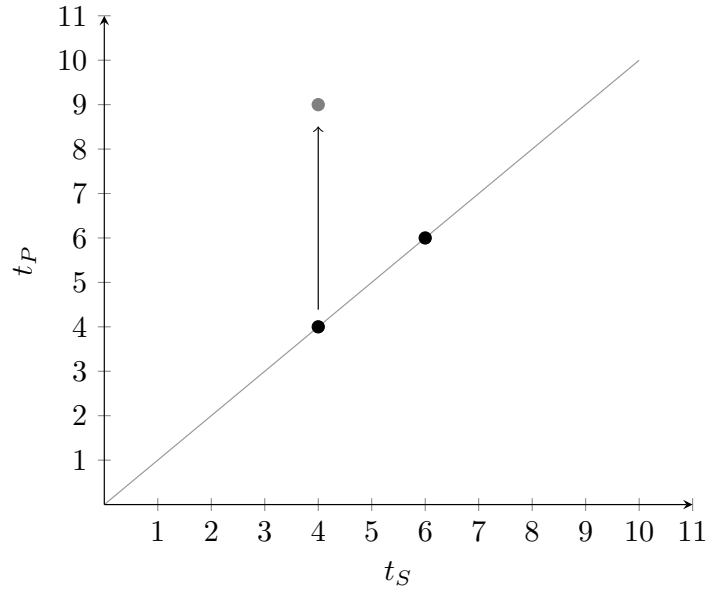


Figure 3.2: Stream ordered by t_S with a predicted tuple

t_P. Predictions use the stream-timestamp t_S from when they were predicted. This way, the order of the stream for t_S is met.

This is illustrated in Figure 3.2. Here, the same stream elements as in Figure 3.1 are drawn, but in a two-dimensional temporal representation. The x-axis represents the stream time t_S and the y-axis the prediction time t_P. The predicted tuple, which was the reason for the unordered stream in Figure 3.1, is now in the correct temporal order according to t_S. The gray diagonal line indicates the placement of the non-predicted tuples: a tuple that is not predicted will be on this line, but not at every point on the line will be a tuple. In this example, two non-predicted tuples are on the line, marked with a dot. When $t_S = t_P$, the tuple is not predicted. When they differ, the tuple is predicted, which is the case with one tuple, marked with a gray dot above the diagonal line.

The extended versions of the stream definitions are called bitemporal, because they have two different temporal dimensions. Definition 3.1 defines the bitemporal version of the logical stream:

Definition 3.1. (Logical bitemporal stream [Bol11]): A logical bitemporal stream S^l of type $\mathcal{T}$ is a potentially infinite multiset (bag) of elements (e, t_S, t_P, n), where $e \in \Omega_{\mathcal{T}}$ is a *tuple* of type $\mathcal{T}$, $t_S \in T$ is the associated *timestamp* in stream-time, $t_P \in T$ is the associated *timestamp* in prediction-time and $n \in \mathbb{N}, n > 0$, denoted the *multiplicity* of the tuple. Let $\mathbb{S}^l_{\mathcal{T}}$ be the set of all logical streams of type $\mathcal{T}$.

The physical bitemporal stream also adds the second time interval for the predicted time:

Definition 3.2. (Physical bitemporal stream): A physical stream S^p of type $\mathcal{T}$ is a potentially infinite, ordered multiset of elements $(e, [t_S, t_{S_E}), \text{Set<}[t_P, t_{P_E})\text{>})$, where $e \in \Omega_{\mathcal{T}}$ is a tuple of type $\mathcal{T}$, and $[t_S, t_{S_E})$ is the half-open time interval for the stream time with $t_S, t_{S_E} \in T$. $[t_P, t_{P_E})$ is a half-open interval for the predicted time with $t_P, t_{P_E} \in T$. One stream element can have multiple, non-overlapping prediction time intervals. A physical stream is non-decreasingly ordered by start timestamps of the stream time t_S.

At this point, a slight difference to the approach from [Bol11] can be seen. While Bolles includes the prediction function into the physical stream, here the physical attributes are extended to temporal types. These temporal types are the prediction functions and deliver non-temporal attribute values for every bitemporal time instance. The difference is due to the moving object algebra, which already defines temporal types and their behavior. This concept is used in this thesis.

Identical to the definition in [Bol11], one stream element can have multiple prediction time intervals. It is possible that, due to the temporal function, a stream element with a temporal type fulfills a predicate at a certain time interval, then, for a few time instances does not fulfill the predicate, and then again fulfills it. Take a moving object as an example, as depicted in Figure 3.3. The object is inside of a certain region between 12:00 and 13:00 o'clock (red line), moves out and then enters the region again (second red line between 14:00 and 15:00 o'clock). A predicate which checks if the moving object is inside that region would have two distinct time intervals as a result.

A solution to deal with such a behavior would be to create multiple stream elements with the same content and the same stream time, but with different prediction time intervals (the times where the moving object is inside the region). Nevertheless, this would introduce a few problems to the stream processing. On the one hand, cost models for query analysis would need to consider that a selection (i. e., filter) operation could create more stream elements than it consumes. On the other hand, element windows would change their behavior. The same element would exist multiple times with different prediction times and therefore could remove parts from itself from an element window. Imagine an element window of size one and a stream element that exists twice with

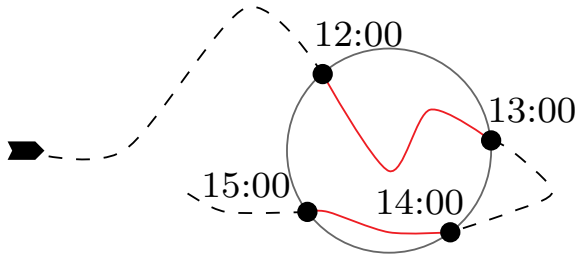


Figure 3.3: A moving vessel being in the queried region twice

different prediction times. The element window would only hold one of the two elements, so the stream element would remove a part from itself from the window, which is typically an unwanted behavior.

3.2 Temporal Types

Güting et al. [GBE+00] define a type system for non-temporal and temporal types, which is the basis for the approach in this thesis. This section describes the logical integration of the type system into a DSMS with the extended time-interval approach explained in Section 3.1.

3.2.1 Naming Scheme

Güting et al. [GBE+00] define and describe non-temporal and temporal types. In the following, we use the naming scheme from Galić et al. [GMKB12] which also adapt the moving object algebra. The temporal versions of a type is written as *temporal([type])* (e. g., *temporal(point)*) or in short as *tpoint*. This maps to the naming scheme from Güting et al. [GBE+00] where it is written as *moving([type])* or in short as *m[type]* with *[type]* as the respective type. Here, *temporal* is used to underline the fact that the approach does not only apply to spatially moving objects but to simple types such as *integer* as well (as is does in [GBE+00], too). For example, when an *integer* value is lifted, it becomes a temporal type called *temporal(integer)* or in short *tinteger*.

3.2.2 Temporal Function

A temporal type consists of a *temporal function* of type α which returns for a given time instance $t \in T$ a value of the type of the non-temporal counterpart:

Definition 3.3. (Temporal Function): A temporal function f_α is a function $f_\alpha : T \rightarrow \alpha$ that for a given time instance $t \in T$ returns a value of the type α.

The temporal function of a temporal type calculates a value of the respective non-temporal version of the type for a given time instance. On the logical level, no temporal types exists: for each single t_P a value of the non-temporal type is available (calculated by the temporal function). Therefore, the operations can be defined on the non-temporal types [GBE+00].

3.3 Operations on Moving Object Streams

The moving object algebra defines operations on the defined spatial and non-spatial types (cf. 2.5). All operations are defined on the non-temporal versions of the types and are then, by a process called *lifting*, transferred to the temporal types. This section describes how to integrate these operations into a DSMS with an interval approach.

3.3.1 General Approach

The moving object algebra [GBE+00] defines many operations such as the set operations "touches" or "in_interior" on non-temporal types and then uses the approach called "lifting" to apply these operations on temporal types or combinations of temporal and non-temporal types (cf. 2.5). This approach can be translated to the bitemporal interval approach in a DSMS.

On the logical view of a data stream, all operations and types are non-temporal. The operations can be performed without temporal considerations as the values of the types are static at the single bitemporal time instance. The lifting happens when applying the operations to physical streams: here, temporal types exist. When applying the operations on temporal types, the results are also stream elements with temporal types with a certain time interval in the prediction time dimension t_P.

In the physical stream, the prediction time interval $[t_P, t_{P_E})$ represents the *periods* value in the moving object algebra, i. e., the sets of intervals or points in time when the temporal function is defined, which also states the periods at which the result is valid, or, in other words, exists. Hence, the collective result for one stream timestamp t_S is exactly the result of the operation in the moving object algebra. This is a fundamental part of the integration of the moving object algebra into the bitemporal time-interval approach: as the stream timestamp t_S does not exist in the moving object algebra, it is not affected by any operations from it. The temporal operations, i. e., the *lifted* operations are represented only by the prediction timestamp t_P. Doing so, the stream algebra by [KS09] is minimal affected, just as defined in [Bol11].

To integrate the operations from the moving object algebra, the operators in the DSMS need to be slightly changed to work with the bitemporal approach. For example, the select and the map operator are extended. The affected operators are explained in the following sections.

Additionally, the moving object algebra defines operations that work directly on temporal types, without a lifting approach. These cannot work on non-temporal types. For example, the operations class "Rate of change" includes "speed" and "turn". These operations work directly on temporal types and cannot be applied to non-temporal types.

An example for such an operation that works, for example, on a temporal integer or temporal real value is the "atMin" operation. It reduces a temporal attribute to those values which are minimal with respect to a given order. For this function to work, the whole temporal attribute is needed, the function works on the temporal type directly and does not need to be lifted to the temporal dimension.

3.3.2 Predicates

Predicates are operations that evaluate to a Boolean value. An example for such an operation is the binary *in_interor* operation. The signature and semantics of this operation are as follows [GBE+00]:

Name:	in_interior
Signature:	$\pi \times \sigma \rightarrow bool$
Semantics:	$u \in U^o$

The *in_interor* operation takes a point (π) and a point set (σ) (e. g., *points*, *line* or *region*) and results in a *bool* value. The semantics is that the predicate checks if the given *point u* is within in the interior of a region which is the *point set* U^o.

Predicates are, for example, used in a selection operator [Krä07]. When using such a predicate in a bitemporal data stream, the result is the set of tuples at which the predicate returns `true`. The definition of the bitemporal select operator shows how the predicate is used to select the stream elements:

Definition 3.4. (Bitemporal Selection [Bol11]): Let S^L be a logical bitemporal data stream and p a predicate over the tuple e. Then, for the selection operator σ, the following holds:

$$\sigma(S^L, p) = \{(e, t_S, t_P, n) | (e, t_S, t_P, n) = r \in S^L \wedge p(r)\}$$

The bitemporal join operator also works with predicates. Its definition can be found in [Bol11].

Lifting

Note that for each stream timestamp t_S an infinite amount of prediction timestamps t_P can exist (resulting in a potentially infinite amount of stream elements for one t_S). Hence, the result of the selection operation "does not state *if*, but *when*—in the sense of the prediction time—the predicate is fulfilled" [Bol11]. This result, the times when the predicate is fulfilled, is the equivalent to the *periods* value in the moving object algebra.

3.3.3 Other Operations

Other operations from the categories “set operations”, “aggregation”, “numeric”, “distance and direction” and “base type specific” from [GBE+00] can be used within the map operator [Krä07]. Here, the same logic as for the predicates apply. The operations are done on the single stream elements in the bitemporal space. Hence, the operations are done in a non-temporal manner and the temporal behavior of the temporal attributes is represented by the second temporal dimension, the prediction time t_P. The mapping operator again can be defined as in [Bol11]:

Definition 3.5. (Bitemporal Mapping [Bol11]): Let f_m be a non-temporal mapping function which can be applied to each stream element separately. Then the mapping operator is defined as follows:

$$\upsilon(S^L, f_m) = \{(\hat{e}, t_S, t_P, n) | \exists (e, t_S, t_P, n) = r \in S^L \wedge f_m(r) = (\hat{e}, t_S, t_P, n)\}$$

3.3.4 Aggregations

Aggregations on data streams aggregate stream elements which are valid at the same time instances. Hence, an aggregation typically reduces the number of data stream elements in a stream. For example, an aggregation could calculate the sum of an attribute of all valid stream elements. The bitemporal aggregation can be defined similar to the aggregation in the interval approach from [Krä07]. The aggregate is calculated using an aggregation function f_{agg} which uses a set of valid stream elements as input. The elements are valid both in the stream time as well as in the prediction time dimension. The aggregation operator can be defined similar to [Krä07]:

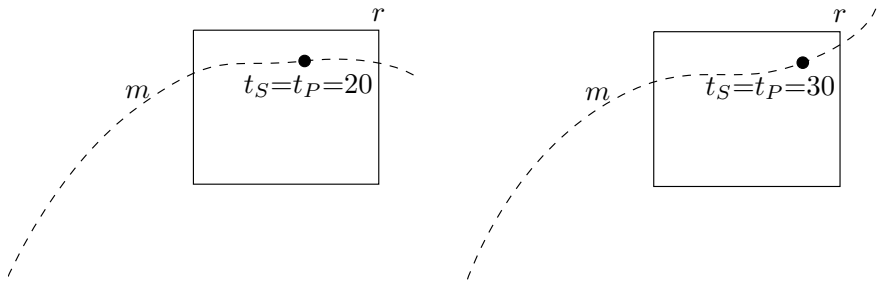
Definition 3.6. Bitemporal Scalar Aggregation (following [Krä07]) Let f_{agg} be a non-temporal aggregation function. Then the aggregation operator is defined as follows:

$\alpha(S^L, f_{agg}) = \{(agg, t_S, t_P, 1) | \exists X \subseteq S^L . X \neq \emptyset \wedge X = \{(e, n) | (e, t_S, t_P, n) \in S^L\} \wedge agg = f_{agg}(X)\}$

3.3.5 Example

To illustrate how moving types and operations connect to the bitemporal time model, Figure 3.4 depicts an example. On each subfigure, two spatial objects are shown: one *tpoint* called *m* and one *region* called *r*. Subfigure 3.4a shows the situation at an earlier point in time than Subfigure 3.4b. Let us assume that the first situation is shown at $t_S = 20$ and the second at $t_S = 30$. In the physical representation, the moving object is a temporal point while in the logical representation, it consists of an infinite amount of non-temporal points in the prediction time domain t_P.

The dashed line in both figures shows the development of the temporal type *m* on the second temporal dimension t_P, the prediction time. The region does not have a



(a) Situation at an early t_S (b) Situation at a later t_S

Figure 3.4: Example of an in_interior operation of a *tpoint* "*m*" and a *region* "*r*"

development because it is not a temporal type. At one point, marked with a dot, the stream time is equal to the prediction time ($t_S = t_P$). The prediction can, dependent on the prediction function, vary between two different stream elements (in stream time), i. e. the dashed line can look different for two different stream timestamps t_S. This can be seen in the figure where the second prediction for the future varies from the prediction in the left figure. The time interval of the prediction time can be from 0 to ∞ if it is not limited. In the figure, the time interval is only drawn partly.

These two spatial types can now be used for a spatial operation, for example, the *in_-interior* operation (see 3.3.2). The non-temporal operation can be applied to every single stream element independently. Hence, for each t_S there is a set of stream elements with different t_P that fulfill the predicate.

In this example, a temporal point and a non-temporal region is given, wherefore the following lifted operation is used:

$tpoint \times region \rightarrow tbool$ **[in_interor]**

Let us assume that the trajectory of m is drawn from left to right (i. e., t_P increases to the right). Let us further assume that on the left subfigure m enters r at $t_P = 10$ and leaves the region at $t_P = 30$. Then, the result of the *in_interior* operation at $t_S = 20$ is that the operations returns `false` for $0 \leq t_P \leq 10$ ($\leq$ instead of $<$ because exactly at the border the point is not inside the region), `true` for $10 < t_P < 30$ and `false` for $30 \leq t_P \leq \infty$. If this predicate would be used for a select operation, the stream elements for which the predicate is `false` would be filtered out so that only the elements within the region would remain in the result set.

In the logical representation, the result would contain a set of 19 stream elements, all at $t_S = 20$ and with 19 different t_P, each containing a *point* (not a *tpoint*). In the physical (lifted) representation, the result would contain one stream element with the interval $[20, 21)$ for the stream time and $[11, 30)$ for the prediction time and a *tpoint* for the location of the moving object.

At $t_S = 30$, the situation has changed. The temporal function for the temporal point is not the same, the moving object has been slower than expected at $t_S = 20$. Now, the entry time is still at $t_P = 10$ but the exit time is at $t_P = 35$. The results of the operation for the stream elements with $t_S = 30$ change accordingly.

3.4 Summary

This chapter has shown that the moving object algebra with its temporal types and lifted operations can be integrated into the logical processing model of a DSMS using a bitemporal time model.

To integrate the concepts of the moving object algebra into an interval-based DSMS, the time model needs to be extended, which is done in Section 3.1. The bitemporal time model introduces a prediction timestamp t_P to the data streams with which multiple stream elements can exist at the same stream timestamp t_S without interrupting the necessary order of elements.

The basic concepts of the moving object algebra are the topic of Section 3.2 and 3.3. Temporal types (cf. 3.2) are functions which can calculate a value of a certain type for a given point in time. This way, the prediction of moving objects to points in time where the exact location is not known can be done. On these types, a number of operations (cf. 3.3) are defined in the moving object algebra. The section describes how the temporal behavior of the operations can be mapped into the streaming model of an interval-based DSMS.

This chapter describes the logical integration. The next chapter will describe the physical integration, which will take bitemporal time-intervals and performance considerations into account.

4 Physical Integration

The logical integration of the moving object algebra with its temporal attributes and lifted operations into a DSMS has been explained in Chapter 3. The physical integration is the topic of this chapter. The logical definitions leave out some practical limitations which need to be considered when integrating the concepts into a real DSMS to process data streams from moving objects.

First, the connection between logical and physical data streams is defined in Section 4.1. This is necessary to apply the definitions from Chapter 3 to physical data streams. The next step in Section 4.2 is the creation of temporal attributes, for example via a transformation from non-temporal attributes such as a *point* to temporal attributes such as a *tpoint*. When having temporal types, the second temporal dimension, the prediction time, comes into consideration. Details on the prediction time are explained in Section 4.3. This includes, among others, the manipulation of the prediction time, for example by the aggregation operation (see Section 4.3.5), and its granularity.

The standard operations on data steams such as aggregations and expressions need to be extended to work in the prediction time dimension with temporal attributes. Expressions that contain temporal attributes are "lifted" to work with the prediction time dimension, which is explained in Section 4.4.

Predictions always come with an uncertainty, which can reduce the trustworthiness of a value in a temporal attribute. How this trust value can be represented is described in Section 4.5.

When querying moving object data streams, more than one moving object is part of a stream. Rather, a number of objects frequently report their location or even stop reporting their location. To query such a data stream, for example with a radius query (i. e., distance query), the locations of the objects need to be combined. Doing this, the temporal dimension of the data makes the combination of data from different moving objects a difficult task. The challenges and possible solutions for this goal are described in Section 4.6.

The filter and refine approach is a typical efficiency improvement step for spatial queries. It filters out elements which are not part of the query result with computationally cheaper calculations than the "real" spatial predicate. The integration of this approach into the query processing on moving object data streams is the topic of Section 4.8.

With these topics covered, this chapter lays the foundation for the implementation of the concept into a DSMS, which is the topic of the next chapter.

4.1 Physical Moving Object Data Stream

A physical moving object data stream is used for the implementation and uses the temporal types (cf. 3.2) to represent moving objects. This representation is used to handle continuous movement with a potentially infinite number of locations with limited resources (e. g., memory).

4.1.1 Transformation

The physical implementation of the concept needs to be semantically equivalent to the logical definition, i. e., the results of the query plans need to be equivalent [KS09]. This can be shown by a transformation from a physical bitemporal stream to a logical bitemporal stream, similar to the transformation defined in Section 2.3.4. The bitemporal transformation is similar to the definition from [Bol11].

Definition 4.1. (Physical Stream To Logical Stream): Let α_t be an attribute of a temporal type which has a temporal function f_α (cf. 3.2). Further, let $t_{temporal}$ be the set of all temporal types. Then, the following function transforms a physical bitemporal stream with temporal attributes to a logical bitemporal stream without temporal attributes.

$$\varphi^{p \mapsto l}(S^p) := \{(\hat{e}, \hat{t}_S, \hat{t}_P, n) \in \Omega_{\mathfrak{T}} \times T \times T \times \mathbb{N} \mid \tag{4.1}$$

$$\exists (e, [t_S, t_{S_E}), \text{Set}<[t_P, t_{P_E})>) \in S^P : \tag{4.2}$$

$$\hat{t}_S \in [t_S, t_{S_E}) \wedge \tag{4.3}$$

$$\exists [t_P, t_{P_E}) \in \text{Set}<[t_P, t_{P_E})> : \hat{t}_P \in [t_P, t_{P_E}) \wedge \tag{4.4}$$

$$\forall \alpha_t \in e : \alpha_t \in t_{temporal} \wedge f_\alpha(\hat{t}_P) \in \hat{e}\} \tag{4.5}$$

Let us take a closer look at this formula. In Line 4.1, the logical stream element is described, which contains a set of attributes ($\hat{e}$), a timestamp in stream time ($\hat{t}_S$), a timestamp in prediction time ($\hat{t}_P$) and a counter to distinguish between equal elements (n). For this logical element a physical element needs to exist in the physical stream (Line 4.2). A stream element in the physical stream also contains a set of attributes (e), an interval for the stream time ($[t_S, t_{S_E})$, the "E" in t_{S_E} stands for "end") and a set of intervals for the prediction time (Set<$[t_P, t_{P_E})$>). Note that in the examples, instead of a set of prediction time intervals, often only one interval is used for simplification.

The parts of the logical stream element are defined by the existence within the physical stream: the stream timestamp $\hat{t}_S$ needs to be within the interval of the stream time in the physical stream (Line 4.3) and the prediction timestamp $\hat{t}_P$ needs to be within one of the prediction time intervals in the Set<$[t_P, t_{P_E})$> (Line 4.4). Additionally, the results of all temporal attributes in the physical stream element for the selected prediction time $\hat{t}_P$ need to be in the attributes of the logical stream element (Line 4.5). In other words,

the temporal functions are solved for the timestamp $\hat{t}_P$ and written to the logical stream element so that no temporal attributes exist in the logical stream.

4.1.2 Example Transformation from Physical to Logical

The definition shows that the conversion from physical to logical contains the stream time, the prediction time and the temporal attributes. While the physical stream contains time intervals and temporal attributes, the logical stream only has points in time and non-temporal attributes.

To illustrate the transformation from a physical to a logical bitemporal data stream with temporal types, Figure 4.1 depicts a physical stream on the left and its logical counterpart on the right. The depicted stream is a moving object stream with the typical attributes: the location of the moving object as a point and the identifier for this object as an integer (int). The location is given as a *tpoint*: the function can predict the location of the object to an arbitrary point in time and does this for the given interval of the prediction time interval t_P (which, for the sake of simplicity, is a single interval in this example, i. e., a set with only one element). The transformation from the physical to the logical stream in this example is done using the transformation function in Definition 4.1.

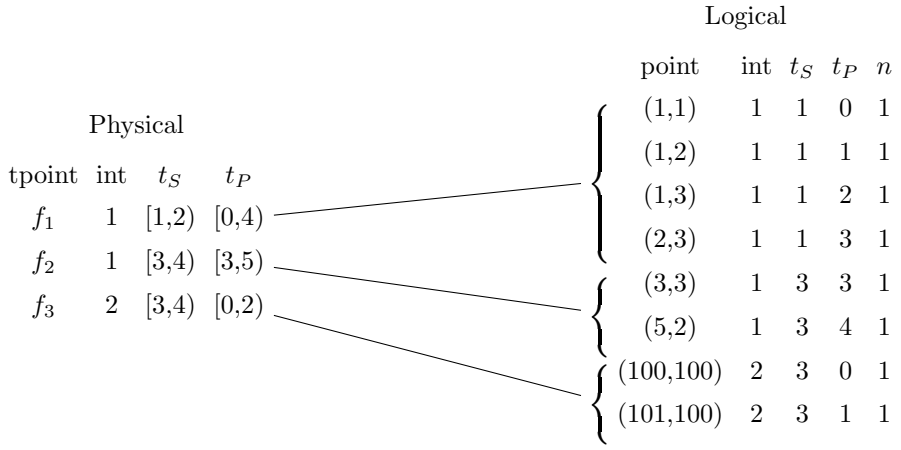


Figure 4.1: Example of a physical data stream and its logical counterpart

Each stream element of the physical stream on the left can translate to multiple elements in the logical stream. With this transformation, it can be shown that the physical implementation is equivalent to the logical semantics. In this example, the first physical tuple is translated to four logical stream elements because the time-interval in the prediction time t_P spans from 0 to 4 (excluded). The temporal function of the temporal point *tpoint* then calculates the location of the moving object with the $id = 1$ for each time instance in the prediction time interval. The id stays the same because it is a non-temporal type. The same procedure is done for the other two stream elements in the physical

stream. Note that the predictions for the same point in time can differ for different t_S. This is useful because the knowledge of the prediction function can change over time what in turn can lead to different predictions than before. For example, when a moving object changes its direction in a way that was not predicted before, the predictions will change, too.

The value of the *tpoint* attribute is different for each stream element. Hence, it can calculate different values for the same t_P. This is the case with the prediction for $t_P = 3$. The temporal function (i. e., the *tpoint*) at $t_S = 1$ predicts the location $(2, 3)$ for this time instance, the temporal function at $t_S = 3$ predicts $(3, 3)$. The third tuple in the physical stream has an earlier t_P. Nevertheless, this is allowed, as the stream is still ordered by the stream timestamp t_S. n is always 1 since there are no equal stream elements in the logical stream in this example. If there would be equal stream elements, n would be used to distinguish them by counting up.

Duration of a Location Update

Moving objects typically move continuously through space, wherefore each location information on a trip of an object is only correct for one point in time. A chronon later, the object will be at a different location. This can be represented with the stream time interval in a data stream, which can be manipulated with window operators. The stream time interval in this case is only one time instance long, i. e., one chronon. Other ways to represent the validity of a location update are possible, for example, to use an element window in which each location update is valid until the next location is available. Nevertheless, to illustrate the gaps between the location updates where the location of the object is not known, a representation with only one chronon validity for each update is chosen here.

This is illustrated in Figure 4.2. A vessel sends its current location three times (at the signal symbols). When received by a DSMS, the location measurement is translated to a tuple. The tuple is only valid one time instant (a chronon) as the location of the moving object is only known at one point in time in the continuous representation (the original measurement). The filled dot depicts the start timestamp t_S and the non-filled dot the end timestamp t_{S_E} of the stream time of a stream element. The location of the moving object in-between these measurements is not known.

4.2 Temporal Attributes

To recap the previous example, temporal attributes can be part of physical data stream elements and are translated to non-temporal attributes in the logical stream. This section is about how to get a temporal attribute into a stream element.

Data sources typically do not know about temporal attributes and deliver non-temporal values. For example, a temperature sensor delivers double values and a data source of

a moving object typically delivers locations, i. e., points. AIS messages, as a source for moving object data, contain the current location in longitude and latitude values, but not a prediction function for the future (or past) trajectory of the vessel. The temporal function from a temporal attribute, which is used to predict the value of that attribute, typically needs to be created by the DSMS itself. There are multiple ways to create a temporal attribute. In general, a function uses a stream element from a physical stream and adds a temporal attribute to the stream elements of that stream:

Definition 4.2. (Temporalization Function): Let β be a temporal attribute, then the following function adds that attribute to a data stream element e.

$$temporalize_{\beta}(e) = \{\hat{e} \mid e \subseteq \hat{e} \wedge \beta \in \hat{e}\}$$

Here, the new temporal attribute is called β and is part of the result stream element $\hat{e}$ after the function $temporalize_{\beta}$ has been applied to e. This definition is very general, as there are multiple ways to insert a temporal attribute to a stream element. One possibility is to use non-temporal attributes and convert them to temporal attributes by creating a temporal function from their current value or history. This approach is described in Section 4.2.1. Another possibility is to receive information about an attribute that can be converted to a temporal function, e. g., a source that itself predicts values of that attribute in the future. This is explained in Section 4.2.2.

4.2.1 DSMS Internal Temporalization

The conversion from a non-temporal attribute to a temporal attribute is called attribute temporalization, or short, temporalization within this thesis. Let us start with an example of a moving object. The moving object sends its location every few minutes in latitude and longitude values. Unfortunately, having this information, it does not tell where it was in-between these messages or where it will be in the future. In short, it does not deliver a temporal function, but only locations for single points in time.

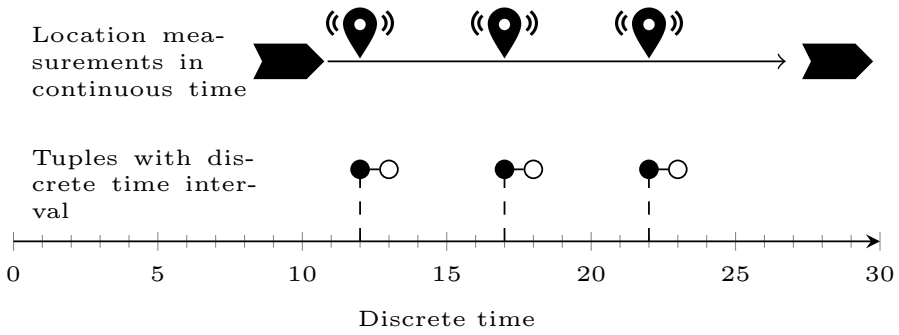


Figure 4.2: Discrete representation of continuous location measurements

time	lat	lng
Vessel 1		
11:35	53.898717	7.334833
11:43	53.91125	7.417233
11:55	53.932683	7.56485
12:10	53.94925	7.668533
Vessel 2		
11:35	53.826	7.59455
11:45	53.8742	7.6076
11:52	53.914267	7.708533
12:00	53.934883	7.810167

Table 4.1: Trajectories from vessel 1 and 2

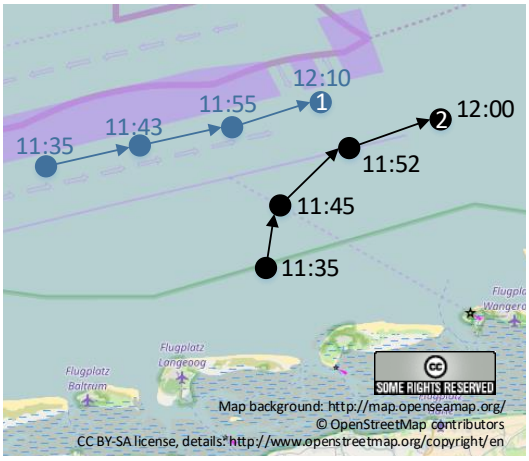


Figure 4.3: Trajectories from vessel 1 and 2 on a map

An example how a data stream of location updates could look like is depicted in Figure 4.3. Here, the trajectories of two vessels (i. e., moving objects) are shown. Table 4.1 contains the raw data. Each moving object (Vessel 1 and Vessel 2) has a list of locations ordered by time. These trajectories are drawn on the map in Figure 4.3. The number (1 and 2) in the last location of each trajectory in Figure 4.3 states the id of the vessel. As the locations of the moving objects in-between the given points are not known, the points are connected by a straight line. The straight line is what a linear prediction function would calculate for the values in-between the known measurements in the second temporal dimension t_P [EGSV99].

Now, let us assume that the latitude and longitude values have been combined to a *point* in the stream element. This attribute shall be made temporal with the temporalization step, i. e., converted to a function which predicts unknown locations.

The process of temporalization is depicted in Figure 4.4. The first step is the transformation of an attribute from a non-temporal type to a temporal type. The moving object data stream in Figure 4.4 has a certain schema, a stream time-interval $[t_S, t_{S_E})$ and a prediction time-interval $[t_P, t_{P_E})$. In this case, the schema consists of a *point* and an *int*. This tuple could be a measurement of the location of a moving object. The location of the moving object is within the *point* value in the schema, the id of the object within the *int* value. The *point* attribute is temporalized in the first step "temporalization of point". With the temporalization operation, the attribute is transformed to its temporal counterpart. In this case, it is transformed from a *point* to a *tpoint*.

Which operator is used to do this temporalization step is not defined here, as it depends on the use case and the necessary data input for the specific temporalization function. Operators that are used for this purpose need to be able to apply functions on the stream

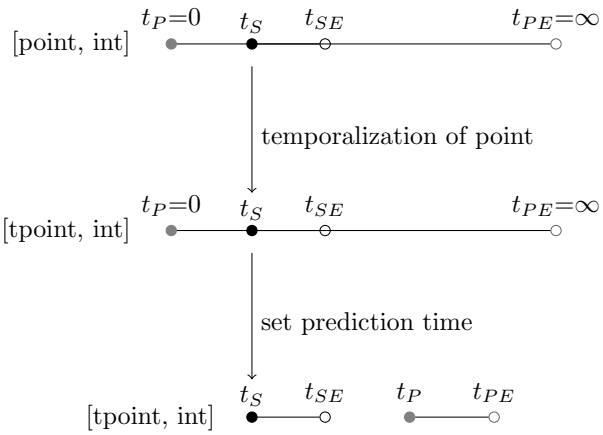


Figure 4.4: Process of temporalization of a physical tuple

elements to make the transition from a non-temporal to a temporal type. The map operator is an operator that applies a map function on a stream element. It is a possible candidate for this purpose if no historical stream data is needed to apply a temporalization function. If the temporalization function needs to use historical data, the aggregation operator is a possible candidate to be extended by new temporalization functions. Hence, the temporalization step can be done with extensions of both of these existing standard operators, depending on the temporalization function.

4.2.2 Temporal Attributes from External Sources

In the previous case, the sensor delivered non-temporal information, i. e., the current location. Other data sources could be capable to deliver more information, for example, about past and possible future values. In that case, the data source also does not directly deliver a temporal attribute, but information on how to create the temporal attribute. This information is written into non-temporal attributes of a data stream element and then used to create a temporal attribute. For example, a navigation system could deliver a list of points where a vehicle will be in the future, together with estimated times when the vehicle will be there. This list could be used to create a temporal function for a temporal point attribute.

Similar to the internal temporalization, the function $temporalize_\beta(e)$ from Definition 4.2 uses the information from the source (e. g., a list with future locations) and applies a function which is able to interpret the data and can transform it to a temporal attribute.

4.3 Prediction Time

The previous sections looked at temporal attributes and how to create them. The prediction time intervals are an important part of a stream element with temporal attributes. They define for which points in time the temporal functions of the temporal attributes need to be calculated, or, from another perspective, can be used. On a semantic level, they define at which time intervals the stream elements are valid in the prediction time dimension.

Both in [Bol11] as well as in [GBE+00], the time-intervals at which a result (e. g., of a predicate in a select operation) is valid is determined by an operation, or a so-called lifted operation in the moving object algebra [GBE+00]. For example, a query could determine the time-interval (or time-intervals) at which a moving point is within a given non-moving region. The result would be a moving point with a temporal function that returns the (predicted) location of that moving point within the time-interval at which the point is within that region (cf. 3.3). The following representation with the second order signature shows this operation:

$tpoint \times region \rightarrow tpoint$ **[in_interior]**

4.3.1 Manual Manipulation of the Prediction Time

An important question is for which time intervals the results needs to be calculated. The times for which the temporal functions are calculated are determined by the prediction time intervals. In some use cases, the user needs to know the results for the next month, in some other cases maybe for the next few seconds or even for the past. Therefore, the prediction time intervals can be chosen by the user when defining the query.

Even though the time intervals can be arbitrarily set, there are some considerations to do before defining the time intervals. First, there are technical limits to the number of possible calculations which need to be considered. Simply calculating all points in time until ∞ can be, depending on the temporal function, impossible. Having a function that can be solved algebraic could reduce the severity of the problem [Bol11], but that is not always the case, e. g., due to temporal functions that are based on complex algorithms such as neural networks.

Second, the use case may not require to predict values that are not in a certain time interval. For example, a use case could only be interested in the next five minutes. Third, predictions always come with an uncertainty, which typically grows with the temporal distance to known values. Predicting a value which is far in the future may lead to only a very limited information gain, because the accuracy of that result is very low.

In other words, the definition of the prediction time intervals needs to fit to the use case and needs to be balanced between completeness of the result and the speed of calculation. If the prediction time intervals are set too short, the query could miss some relevant

results. If the prediction time intervals are too long, the computation could increase the latency of the results, wherefore results could be calculated too late. Another factor for this consideration is the prediction time granularity, which is explained in Section 4.3.3.

To define the query in a way that fits the use case, the time-interval can and should be defined by the query. For example, the time-interval could be reduced to an interval starting at t_S and ending five minutes ahead of t_S. This way, the query "looks" five minutes into the future and reduces the number of points in time to a finite set. In other words, the approach is limited from the question "What is the result of this query?" to "What is the result of this query in the given time interval?". The time interval could be further reduced to a single point in time by setting t_P to a chronon. In that case, the question is further limited: "What is the result of this query at this point in time?".

To achieve this, the predicted time is limited by an operation on the stream element before the temporal operation. As depicted in Figure 4.4, in the non-temporal and the temporalized version of the stream element, the prediction time-interval is $[0, \infty)$. This is the most general possible time-interval as it represents the logical definition of the operations on temporal types. In this example, it is limited to a short time-interval in the future (i. e., ahead of t_S).

The prediction time-interval can be manipulated with an operator similar to a mapping operator:

Definition 4.3. (Set Prediction Time Operator): Let f_p be a function that determines for a given set of attributes and the stream timestamp a prediction timestamp.

$$\nu_{f_p}(S^L) := \{(e, t_S, \hat{t}_P, n) \mid \exists (e, t_S, t_P, n) \in S^L \wedge f_p(e, t_S) = \hat{t}_P\}$$

4.3.2 Prediction Time Manipulation with the Select Operation

The step described in the previous section is the preparation for the rest of the query and should not be done again after an operation on temporal attributes, for example, after a select operation. That is because other operations use the prediction time for their results. The said select operation limits the prediction time intervals to those where the predicate of the select operation is true. If there would be another manual manipulation of the prediction time intervals afterwards, these results would be lost or at least change their semantic meaning.

Manipulating the prediction time first and then applying temporal operations on the stream is compatible to the logic of the bitemporal data stream. Doing it in that order, only the given prediction time-interval is used for operations on temporal types, not all possible time instances. To illustrate that, let us consider a select operation on the bitemporal data stream from the example in Section 3.3.5 with Figure 3.4a. A select operation with the predicate *in_interior*(m, r) would return the following physical stream element: $(m, [20, 21), < [11, 30) >)$ of the schema (tpoint, $[t_S, t_{S_E}), < [t_P, t_{P_E}) >$).

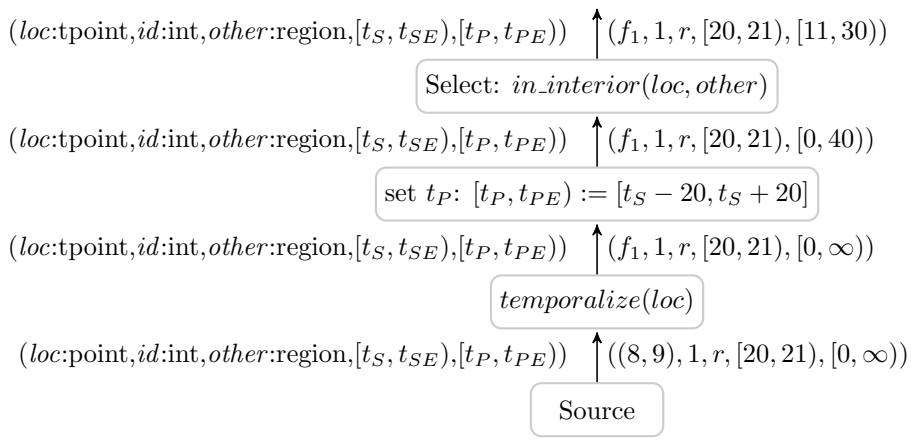


Figure 4.5: Example of a query plan that transforms a point to a temporal point

If another select operation with the inverted predicate *not*(*in_interior*(m, r)) is done on the result of the first select operation, the result should be empty. That is because if the predicate would still consider all possible prediction timestamps (i. e., a prediction time of $[0, \infty)$), the result would be inverted, resulting in a stream element with two prediction time intervals: $(m, [20, 21), < [0, 11), [31, \infty) >)$. Nevertheless, the set of results from the first select does not contain these prediction time intervals, wherefore the prediction time cannot be ignored.

The example from above can be depicted as a query plan. Figure 4.5 shows such a plan. The operators are in the boxes. These perform the operations on the tuples on the right. The schema is depicted on the left. The schema of the incoming stream element (bottom) contains a location of a moving object as a point (*loc*), an identifier of the moving object as an integer (*id*) and a region (*other*). The content of the region here is abbreviated with *r* but could, for example, be a set of points defining the border of the region.

The second operation takes a tuple and transforms an attribute, in this case the *loc* attribute, to a temporal attribute (for details about the temporalization step, see 4.2.1). In this example, a *point* is transformed to a *tpoint*. The content of the stream element changes from a non-temporal point (8,9) to a function which represents the movement over time (f_1). The next step is to set the prediction time to make it fit to the use case and to limit the computational load of this query. Here, the prediction time is simply limited to 20 time instances before and after the stream start timestamp t_S. The last operator performs a selection based on the predicate *in_interior*($loc, other$). The result is a stream element for which the prediction time is limited to the interval in which the object is inside the region.

4.3.3 Prediction Time Granularity

The previous examples worked with stream and prediction time intervals, but never stated how much time is between these integer timestamps. It could be a second or an hour. For the concept it does not matter how much time lies in-between the timestamps. When applying the concept to real queries, the granularity of the time intervals need to be considered and has an impact on the quality of the results and on the computational load for a query. Therefore, the granularity needs to be defined according to the use case.

A time interval in a physical stream is discrete, i. e., consists of a finite number of time instances between the start and the end timestamp. The number of time instances within, for example, one minute, depends on the granularity of the time interval. The granularity could be set to one millisecond, resulting in 60 000 time instances within one minute. If set to one second, only 60 time instances are within the same time interval.

Both the stream time intervals and the prediction time intervals have a certain granularity. The granularity of the prediction time interval can be set independently from the granularity of the stream time. Different granularities increase the flexibility of the stream processing and allow the handling of bigger time intervals for prediction. In other words, when having a long time interval, the computational load can be reduced by choosing a lower granularity for the prediction. Depending on the data and the query, one prediction per second could be sufficient. In contrast to a granularity of a millisecond, the number of necessary predictions is reduced by a factor of 1000.

Concluding, the granularity of the prediction time intervals is another point of flexibility for the user to adapt the query to the needs of the use case.

4.3.4 Number of Prediction Time Intervals

For each data stream element, exactly one list of prediction time intervals exist. Whether it has none, one or even multiple temporal attributes, for all attributes in a data stream element the same prediction time intervals apply. This rises the question whether it could be useful to have a list of prediction time intervals for each temporal attribute. Having multiple prediction times, a stream element could, for example, include two temporal points from two moving objects. Both could be valid at different points in time. Even though this seems to be a legitimate property of the temporal attributes on the first sight, it raises some issues when applying operations on these temporal attributes.

Let us consider a stream element with three temporal attributes:

movement (tpoint)	speed (treal)	temperature (treal)	stream time
some value + [0,10)	some value + [0,10)	some value + [100,200)	[0,1)

This could be a data stream from a runner with its location as a temporal point (*movement*), the *speed* of the runner and the *temperature* of the environment. From such a stream, some could, for example, see if the temperature has an impact on the speed of the runner. Nevertheless, instead of having one prediction time interval, each temporal attribute has its own temporal interval (written as " + [0,10)"). While the movement and the speed are predicted to the same time interval, the temperature is predicted to a completely different time interval way further into the future (pretending that "now" is at point in time 0). On the prediction time dimension, the temperature attribute seems to be unrelated to the other two attributes. This lost relation between the attributes on the prediction time dimension can lead to confusion when applying data stream operations on such a stream.

This problem can be illustrated when applying a select operation on the example stream element. When using a predicate `speed > 10km/h` to limit the stream to all stream elements and prediction times when the runner is faster than 10 km/h, the resulting stream element could look like this:

movement (tpoint)	speed (treal)	temperature (treal)	stream time
some value + [0,10)	some value + [5,8)	some value + [100,200)	[0,1)

Now, the speed is limited to the three points in time (in the prediction time dimension) 5, 6 and 7 where the runner is faster than 10 km/h. As the attributes in one stream element are related to each other, some would now expect to also have the trajectory (i. e., temporal point in the movement attribute) where the runner is that fast and also to know the temperature of these parts of the movement. Unfortunately, this is not possible with this approach. The temporal point is not changed as it has its own prediction time interval and the temperature is not even known during the resulting time interval.

When having only one set of prediction time intervals per stream element, the scenario is solved correctly. The starting stream element is the following:

movement (tpoint)	speed (treal)	temperature (treal)	stream time	prediction time
some value	some value	some value	[0,1)	[0,10)

Here, all three temporal attributes have the same prediction time interval [0,10). When now using the previously described select operation, the result is the following:

movement (tpoint)	speed (treal)	temperature (treal)	stream time	prediction time
some value	some value	some value	[0,1)	[5,8)

Now, the movement of the runner, the speed and the temperature of the environment fit together. It is the same as with the stream time interval which is defined by a window

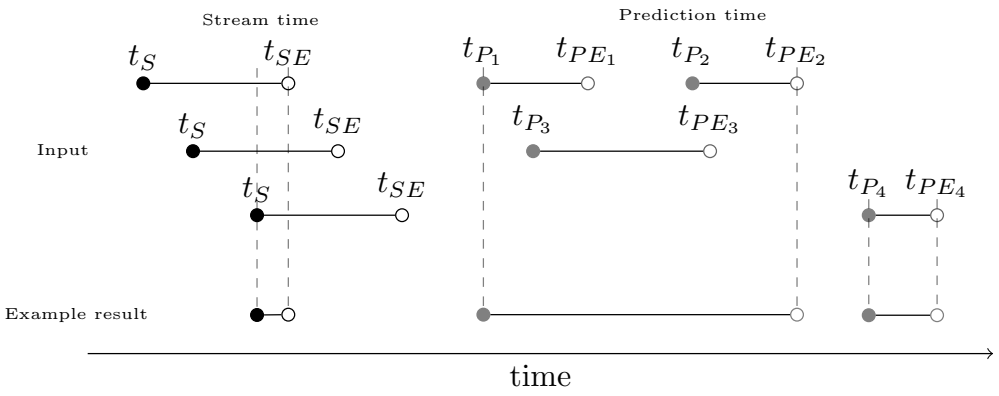


Figure 4.6: Calculation of prediction time for an aggregation with the union merging function

operator. The time interval is used per stream element, not per attribute. When a stream element gets invalid from the stream time interval, the whole stream element is removed from the stream, not some attributes.

4.3.5 Temporal Aggregation

The implications of the select operation on the prediction times has already been described (cf. 4.3.2). Other operations on a bitemporal stream also manipulate the prediction time intervals. Among them is the aggregation operation.

Temporal attributes can be used within aggregation functions. For example, a sum over a temporal attribute from the last stream elements can be calculated. The prediction time interval needs to be manipulated accordingly. In short, the time intervals of the prediction times of the current stream elements (in the window) are unified. This is shown in Figure 4.6.

The current stream elements are depicted by the three input time intervals on the upper left. The stream time intervals of the elements are overlapping. The prediction time intervals, i. e., the second temporal dimension, are right of the stream time intervals, i. e., they are in the future.

Calculating the aggregates would generate multiple results, because each different overlapping situation (only first element is valid, first and second elements are valid, etc.) has its own result. One of these is depicted below the input stream elements: the result for the time span where all three input stream elements are valid. The stream time interval is the intersection of all three time intervals.

In the second temporal dimension (the prediction time) a result for the aggregation function is calculated for each point in the prediction time intervals (cf. 3.3.4). A value

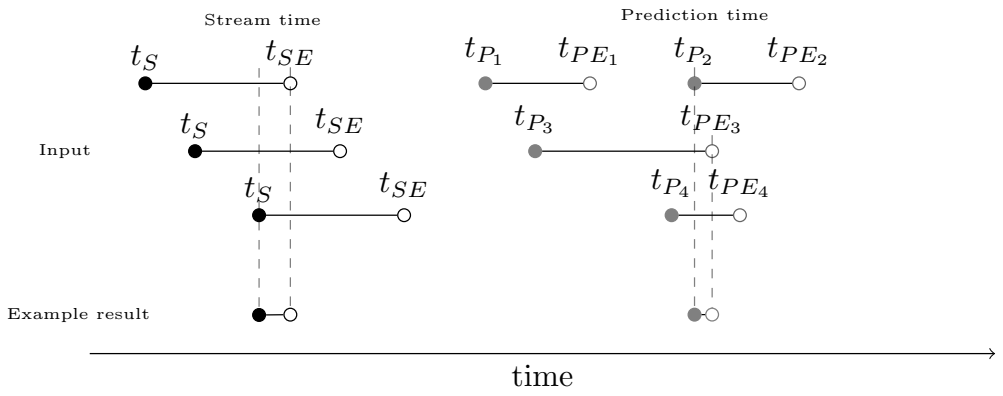


Figure 4.7: Calculation of prediction time for an aggregation with intersection merging function

of a temporal attribute is only used if its prediction time interval is valid at that point in time. In Figure 4.6, the result of the aggregation at t_{P_1} is only calculated using the temporal attribute from the first stream element, because the second and third stream elements are not valid in the prediction time dimension at that point in time. Another example: at time t_{P_3}, the result is calculated from the first and second stream element, as they are both valid in the prediction time dimension at that time.

The result is that the prediction time intervals are merged using the union function, as can be seen in Figure 4.6 at the bottom: the resulting set of prediction time intervals is the union of the prediction time intervals of the three input elements.

Nevertheless, at this point, the user should have the choice to pick the best merging function for the prediction times for the respective use case. The union may not be the desired result for all aggregation operations. The merging function for the prediction time intervals could also be the intersection instead of the union. In that case, the aggregation has only results where all temporal attributes are valid. In the given example in Figure 4.6, the resulting prediction time intervals would be empty.

The slightly changed example in Figure 4.7 shows the result of an intersection merging function for the prediction time intervals. Here, the result set of prediction time intervals is not empty.

Both merging functions for the prediction time in an aggregation operation are possible. Hence, the decision which to use depends on the use case. When working with the union merge function, the result of the aggregation for some points in time in the prediction time dimension ignores `null` values from some stream elements, which, at other points in time, contribute to the result. When using the intersection merge function, the result for the aggregation is at all points in the prediction time created from all stream elements.

Example

The decision which merging function to apply depends on the use case. The following examples give an idea of possible criteria for this decision.

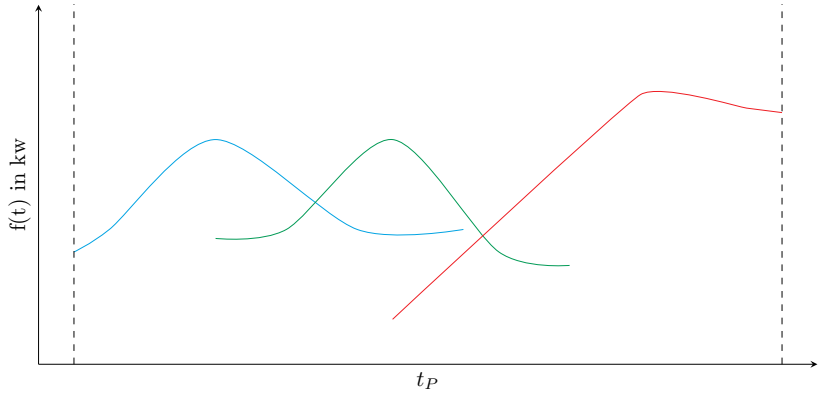


Figure 4.8: Electricity generation by three distinct generators (e. g., solar panels)

The first use case is depicted in Figure 4.8. The power output of three power generators, for example three solar panels, is depicted here. This is the predicted power output from a temporal double attribute in three stream elements. The prediction functions only predict the generated power for certain time periods. The aggregation function is the sum function, calculating the total power generation of all three power generators. The power output shall be calculated no matter from how many power generators a prediction function is available, to have a prediction closest to the real generation as possible. In this case it is better to have a potentially incomplete prediction than to have no prediction for some points in time. Hence, the union merge function is applied in this use case. The aggregation is calculated for all points in time from the start of the first (blue) curve to the end of the last (red) curve. The resulting time span is depicted by the dashed lines.

The second use case is depicted in Figure 4.9. Here, the stock prices from two companies are plotted over time. Again, these values are created from two prediction functions from two stream elements that are currently in an aggregation function. Let the use case be that the “min” aggregation function is applied to get the minimum valued company for each point in the prediction time intervals. To avoid wrong claims about the minimum valued company, the intersection merge function is used in this case. The results are only calculated for the time interval between the dashed lines. If the union merge would be used, the blue “Company 1” would be the result for the other points in time, only because the prediction function for “Company 2” does not provide values for these time intervals.

In conclusion, the merge function needs to be chosen depending on the use case. While the union merge also creates results for time periods where only parts of the

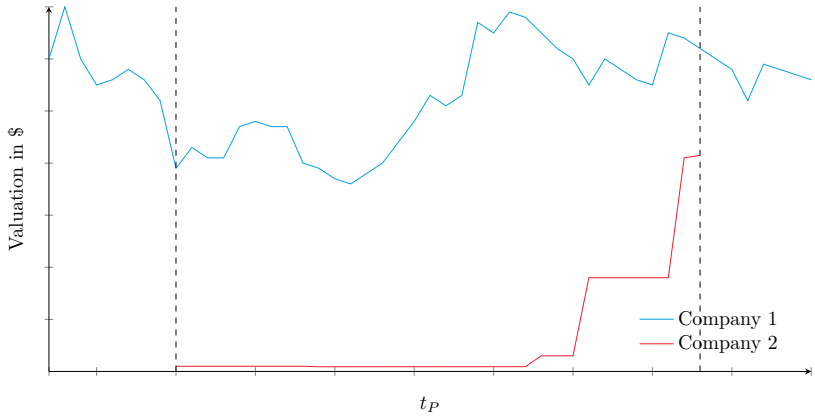


Figure 4.9: Stock market development of two shares

temporal attributes can be used due to their prediction time intervals, the intersection merge only creates results for time periods where all current temporal attributes can be used.

4.3.6 Temporal Join

Similar to the aggregation operation in the previous section, a join operation on a bitemoral stream also implies changes to the prediction time intervals, i. e., the prediction times need to be joined correctly. For the aggregation, two possible merging functions for the prediction time intervals have been presented (cf. Section 4.3.5). For the join, the intersection merging function is applied (cf. Figures 4.7 and 4.9).

When joining two stream elements, both stream elements possibly have temporal attributes. The prediction time intervals on each stream element define when these temporal attributes can be used. The points in time which are not covered cannot be used for any kind of operation on these temporal attributes, e. g., for expressions. Therefore, when joining two stream elements, the prediction time intervals of the joined stream elements cannot cover any points in time where at least one of the temporal attributes cannot be used. This makes it infeasible to union the prediction time intervals of both stream elements, they need to be merged using the intersection.

The following example makes the need for the intersection merge function more clear. Let us consider a stream with a temporal point which represents a moving object. A select operation reduces the prediction time interval to the times where the moving object is within a certain area. For a stream element A that may be from 12:00 o'clock to 13:00 o'clock, i. e. `[12:00,13:00)`, and for a stream element B from 12:30 to 13:30, i. e. `[12:30,13:30)`. When joining these stream elements, a union merge function for the prediction times would result in a prediction time interval `[12:00, 13:30)`. Having

this result, the semantics of the temporal attributes saying that the moving objects are within a certain region would be nullified. Contradictory to the previous result, the moving object from stream element *A* would be said to be within the region up until 13:30 o'clock, which is not the case.

Using the intersection, the semantics stays correct. When joining the two stream elements, the resulting prediction time interval would be `[12:30, 13:00)`. For both moving objects from the original stream elements *A* and *B* the predicate from the previous select operation is still true for all points in time in the prediction time interval: they are within the given region during this time interval. When using the resulting stream element for further operations, e. g., an expression which calculates the distance between these two moving objects, none of the two temporal attributes would return an invalid (e. g., `null`) value for any of the given points in time.

4.4 Lifted Expressions

Expressions are a fundamental part of query processing. They use the attributes from stream elements to calculate their results, which can be written into the result stream element or used as predicates for select and join operations. With the "lifting" concept [GBE+00], standard non-temporal expressions can be applied to temporal attributes. For example, the non-temporal "+" operation can be applied to integer and real values, both non-temporal as well as temporal. Nevertheless, the "+" operation itself does not know about temporal types, as is was developed for non-temporal number values (integers, doubles, ...).

To make it possible for this non-temporal operation to be applied to temporal attributes, the expression is calculated for each point in time in the prediction time intervals with the respective values from the temporal attributes. In other words, for each point in time, the temporal attributes calculate their non-temporal value. This value is used to calculate a non-temporal result with a non-temporal operation (e. g., the "+" operation). Then, the results for the different points in time are combined again to a new temporal attribute, which is the result of the lifted operation.

The following example shows the process when a non-temporal operation is used with a temporal attribute. The stream element has a temporal integer attribute "v" (value). The stream time interval is [0,2) (but is irrelevant here, because it does not affect this example). As mention before, the stream time is not manipulated by temporal operations. The prediction time interval is [5,10) and is important for the operation. The temporal integer has five values, one for each time instance of the prediction time, accordingly. This could be a function or a map from a time to an integer value. For simplicity, in this example a map is used. The resulting stream element can be written as follows:

v (tinteger)	stream time	prediction time
5→10, 6→10, 7→20, 8→40, 9→50	[0,2)	[5,10)

On this element, a non-temporal "+" operation can be applied via a map operator. This is the same "+" operation which is used for non-temporal integer or real values. Let the operation be "v + 5". When the "+" operation would get passed the temporal integer, it would not be able to do anything with it, as it is an unknown type for this operation. Some would need to create a special temporal "+" operation, which is not desired, as each operation would need to be recreated. What happens instead is that five non-temporal stream elements are created, one for each point in the prediction time interval. Hence, the "+" operation works on these tuples:

v (integer)	stream time
10	[0,2)
10	[0,2)
20	[0,2)
40	[0,2)
50	[0,2)

These stream elements never appear in the data stream, they are only created for the non-temporal "+" operation. On these elements, the operation can calculate its results:

v (integer)	stream time
15	[0,2)
15	[0,2)
25	[0,2)
45	[0,2)
55	[0,2)

Again, these stream elements never appear in the stream. They are packed into a temporal attribute, hence, the result looks like the following:

v (tinteger)	stream time	prediction time
5→15, 6→15, 7→25, 8→45, 9→55	[0,2)	[5,10)

This result can again be used to do other operations, for example, a boolean expression "$v > 20$". The process would be the same.

4.4.1 Temporal Operations

The moving object algebra also defines functions for expressions which are not lifted, but work directly on temporal types (see 3.3.1). These functions do not need the non-temporal versions of the stream element explained above, but use the temporal version. For example, the "atMin" function reduces the values of a temporal attribute to those

values which are minimal. Using the result of the example from above, the result of the "atMin(v)" mapping operation would be the following:

v (tinteger)	stream time	prediction time
5→15, 6→15	[0,2)	[5,7)

The temporal integer "v" has been reduced to the points in time where the value is "15" and the prediction time has been limited accordingly.

4.5 Temporal Trust Value

A temporal function is a function from a point in time to a value of a certain type. The function is, for example, used to predict values at points in time when a measured value is not available. The predicted values allow calculating predicted results with it, but the results still stay predictions, which can be inaccurate. When working with the results of a query, it can be useful to know how trustworthy a result is [Bra17]. If a result is very trustworthy, the user can probably rely on this result, if it is not trustworthy, the user maybe does not want to use it for some decisions. To decide this, the trustworthiness of a prediction needs to be known to the user, but typically it is hidden. With the trust value described here, the trustworthiness of a prediction is aimed to be made available to the user.

Following the naming from [BGA17], this meta attribute can be called "trust" and represents how trustworthy a data stream element, or in this case a specific prediction, is. In contrast to [BGA17], a trust value is not only "trustworthy" or "untrustworthy", but can be represented with a continuous real value between 1.0 and 0.0. 1.0 means that the result is considered as correct, while 0.0 means that the results is considered as incorrect.

Nevertheless, to provide the trustworthiness or rather accuracy of a temporal attribute, a single trust value is not sufficient, because the accuracy of the prediction can change over time. For example, if a prediction is close to known values, it is probably more accurate (and with that, trustworthy) than if it is far from a known value (in the temporal dimension). Therefore, the prediction trustworthiness cannot be represented by a single value but needs to be known for each point in the prediction time interval. Hence, the prediction accuracy itself is a temporal (meta) attribute that has for each point in the prediction time interval a trust value. In other words, the trust value for a temporal attribute is itself a temporal real attribute.

Figure 4.10 gives an example of a trust function. At the points in time (the x-axis) where the exact value of the temporal attribute is known, the trust is at its highest value, marked with a dot. In-between, before and after the known points, the trust value decreases. Following operations and applications can use this value to interpret the results, e. g., to increase safety margins. In addition to just using the trust value to interpret the

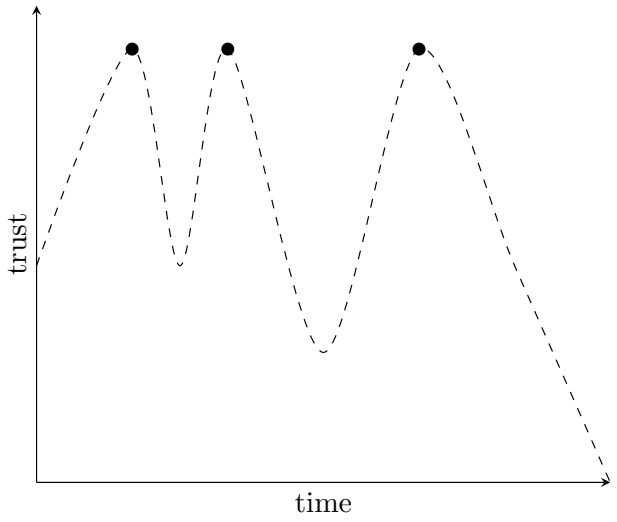


Figure 4.10: Trust value over time

results, they can also be used within the queries. For example, when having a radius query around a (predicted) location of a moving object, the radius could be increased or decreased depending on the trust of the temporal spatial point, i. e., the location of the moving object over time.

Note that the trust function, i. e., the temporal double, is itself only a prediction function which can be inaccurate. The estimated trust of a value of a temporal attribute can be wrong and, for example, estimate a high trust where the prediction is very inaccurate. As the trust value is also a temporal attribute, it can also have a trust value for itself, i. e., a value stating how trustworthy the trust is. Concept-wise this can go on and on, but for implementation purposes, at some point, there should be a function which states a static, e. g., `1.0` trust.

Merging Trust Values

Each temporal attribute can have a trust value. When combining multiple temporal attributes, for example through an expression (e. g., `tempAttr1 * tempAttr2`) or with an aggregation operation, the trust attributes need to be merged. In that case, for each point in the prediction time the minimum trust value of all involved temporal attributes is used. That follows the merge function in [BGA17], where the trust of multiple stream elements is always the minimal available trust. That is because a high trust of a certain temporal attribute cannot increase the trust of a different temporal attribute at a certain point in time. In short, by using the minimal trust value, the trust states a pessimistic trust estimation.

A simplification of this concept where each temporal attribute has a trust value is a variant where only one trust value per stream element exists. This simplifies the concept,

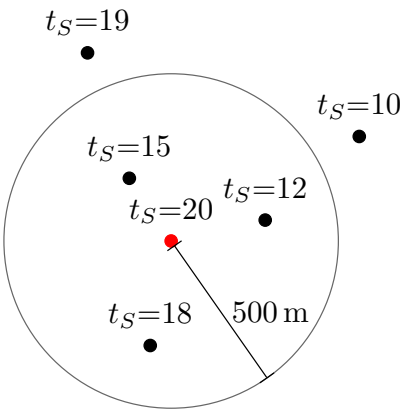


Figure 4.11: Center moving object at $t_S = 20$ with surrounding moving objects

reduces the overhead for temporal attributes, but reduces the accuracy of the trust value. That is because two temporal attributes in the same stream element can have a different accuracy for the same points in time in the prediction time dimension. Whether a single trust value is used or one trust value for each temporal attribute, the merge function stays the same.

This approach to represent a trust value and work with uncertainties and probabilities is rather simple but shows the advantages of using a DSMS that has mechanisms for handling metadata. Different trust functions for certain use cases as well as different merge functions to merge trust values from different stream elements or temporal attributes can be applied. More advanced concepts for handling probabilities in data streams can be found in [Kuk15].

4.6 Queries with Multiple Moving Objects

The physical concept of the temporal types has been described in the previous sections of this chapter. Beginning with this section, another challenge that comes with the query design for multiple moving objects is tackled. It appears independently from the temporal types, nevertheless, it is illustrated with examples containing temporal types.

In a previous example in Section 4.3.2, a query with a select operation over the distance between two spatial objects contains one temporal object (*loc*) and one non-temporal object (*other*). The temporal object is updated when new stream elements arrive, the non-temporal object is not updated but static and will not change over time. In such a situation, the data stream with the locations of the moving objects can be easily enriched with the static object to calculate the results of spatial operations. In other words, each moving object can be calculated independently in this query. The situation changes when the operations are done on multiple moving objects which change with new elements in the stream.

Consider the situation depicted in Figure 4.11: six moving objects irregularly send their current location to the DSMS. Their location updates are not synchronized. The user wants to know which moving objects are within a distance of 500 meters around the moving object with the $id = 1$ (the "center element"). This is called a radius or distance [Bri07] query. The predicate for a select operator that does such a radius query can use the *distance* operation from the moving object algebra: $distance(loc1, loc2) < 500m$. Here, *loc1* would be the location from the moving object with the $id = 1$ and *loc2* would be the location of the other moving object. Both locations would be of type *tpoint*. The result is a time interval in prediction time in which the predicate is fulfilled. When the user wants to know the objects that are close to the center object "now", the prediction time could be set to a single point in time exactly at the stream time t_S of the newest element.

4.6.1 Trigger

An important consideration is the question when to calculate new results for the query, i. e., which stream elements trigger which new output. In a database, a radius query is run once and the output reflects the known situation at the time the query is executed. In a streaming system, the data that is available to run the query changes continuously. Hence, it needs to be defined when new results have to be calculated. For queries that observe a certain moving object, e. g. a radius query, the following triggers are possible:

1. on each center element
2. on each non-center element
3. both (on each element)

In the first case, with a calculation on each new stream element from the center object, this element is combined with the newest element of each other known moving object. As the result, a "picture" of the whole situation at the time instance of the newest stream element of the center element is calculated. A disadvantage of this approach is that the time instances and situations in-between the location updates of the center elements are not calculated. If the direction of another moving object changes or a new moving object appears, results can be missing. The advantage is that less calculations are necessary.

A scenario which shows the consequences of such a configuration is depicted in Figure 4.12. For the sake of simplicity, let us consider that the moving objects are of the type *tpoint* but that the prediction always returns the last known value (i. e., the objects are "jumping"). There are three subsequent location updates depicted. At each time instance, the newest locations of all moving objects are known (i. e., within the window of the query plan). At the left figure, the center element has updated its location at $t_S = 20$. The query calculates a result at this time instance and the user knows the result of the radius query at this point in time (three other moving objects are in the circle). The

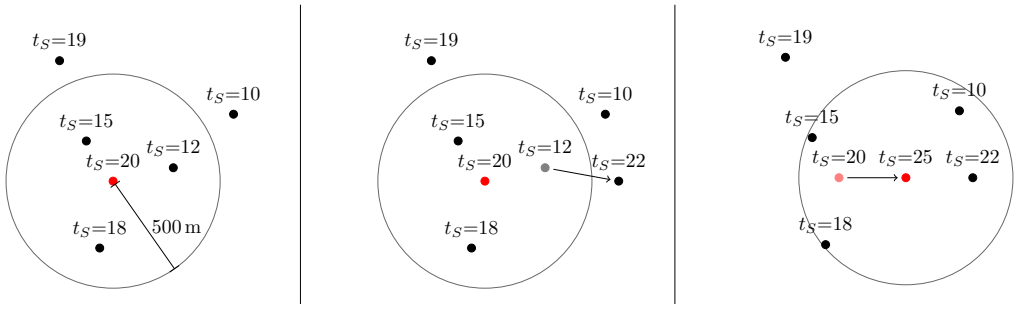


Figure 4.12: Scenario with moving objects over time

next incoming data stream element (i. e., location update) is at $t_S = 22$ from a different moving object, which moves out of the radius. In this configuration, this update does not trigger a new result. Hence, this situation with only two elements in the radius is missed. The third situation at $t_S = 25$ is again triggered by the center element and therefore creates a query result (with four elements in the radius).

The second case calculates a result for each other stream element than the center element. When another moving object has a location update, this stream element is combined with the newest known element from the center element. The result of the query (e. g., a radius query) is calculated step by step with the location updates, not all at once.

This approach again can be seen in Figure 4.12. As with the previous approach, results can be missing. When the center element sends an update, no update is triggered with this approach (the left and the right situation depicted). In contrast to the previous configuration, now the situation in the middle triggers a distance calculation between the center element and the updated element at $t_S = 22$. The distance is greater than the given radius of 500 m so that at $t_S = 22$ it is known that this element is no longer within the radius.

To get all results, both approaches need to be combined, hence, new results need to be calculated at every new location update from a moving object. This comes at the price of more necessary calculations and therefore possibly a poorer query performance. The decision, which approach is used, can be made by the user. On a technical level, the approaches can be implemented with combinations of standard DSMS operators such as selections, windows and joins.

4.6.2 Sample Radius Query

Concluding from these considerations, for a binary spatial operation (such as a distance calculation), a data stream is needed with the following schema: *(tpoint, int, tpoint,*

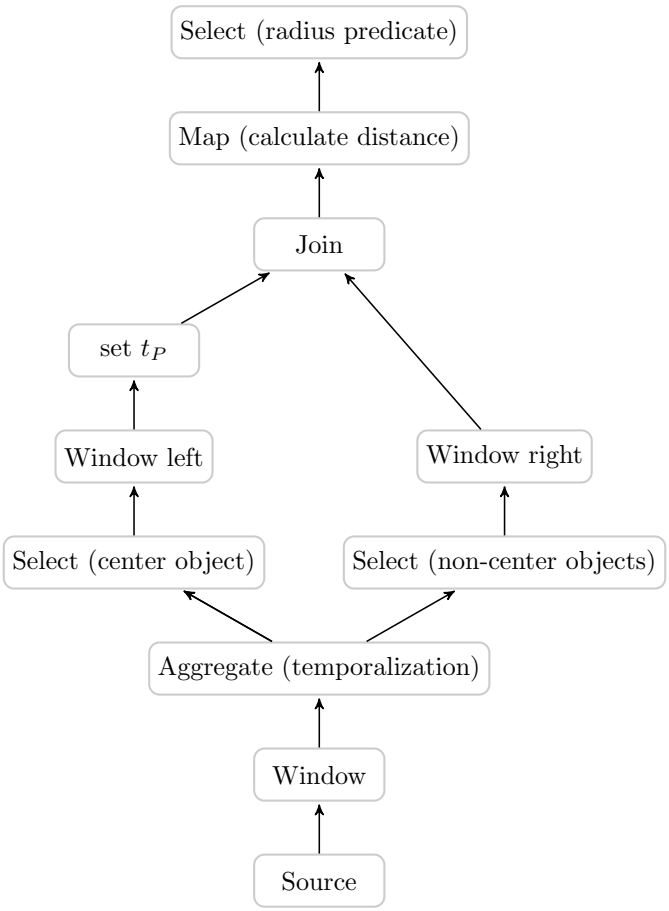


Figure 4.13: Query plan for a radius query

int, $[t_S, t_{S_E})$*,* $[t_P, t_{P_E}))$. For queries that calculate the situation only at the one time instance that marks the newest known time ("now"), the stream time interval needs to be a chronon and the prediction time interval needs to be equal to the stream time interval. The *int* values are for the identification numbers of the moving objects. Unfortunately, the incoming stream of location updates has a different schema: *(point, int,* $[t_S, t_{S_E})$*,* $[t_P, t_{P_E}))$.

A number of steps are necessary to achieve the described schema for two combined location updates. First, the *point* attribute of all stream elements need to be temporalized. Second, the prediction time of the center object needs to be set to the stream time. Third, the center object needs to be joined with the newest stream element of each other moving object. The last step is the distance predicate in a select operator. A possible query plan to achieve these steps can be seen in Figure 4.13.

4.6.3 Step 1: Temporalization

The temporalization is done as defined in Section 4.2.1. In this case, an aggregate function is used, because the history of the respective moving object is needed to create the temporal function of the temporal type (*tpoint*). To prevent an overflow of the data that the aggregate operator keeps in memory and to emphasize newer data in the prediction rather than old data, a window is used to reduce the amount of stream elements used for the aggregation step. The window operator manipulates the end timestamp of the stream time (t_{S_E}) but does not change the prediction time. For example, a time window with the size of one hour could be used.

4.6.4 Step 2: Self-Join with Center Object

The next step is the self-join with the center object. Here, the center object (on the left) is processed in a different way than the other objects (on the right). To separate the elements, two select operators with an inverted predicate are used, e. g., $id = 1$ on the left and $id \neq 1$ on the right.

The window operators above of the select operators control the trigger behavior explained above. Using an element window with only the newest element per moving object (size set to 1 and partitioned by the *id*) on either side, a new element on the other side triggers a combination with all elements (which are currently in the window). This results in the following combinations for the described behavior.

1. Trigger on each center element: chronon window on the left, 1-element window on the right
2. Trigger on each other element: 1-element window on the left, chronon window on the right.
3. Trigger on each element: 1-element window on both sides.

After the window operator of the center element, the next operator manipulates the prediction time (set t_P). It is changed from the original $[0, \infty)$ to only a shorter time period, and, following the example description from above, to only one time instance, which marks exactly the "current time" or "now": $t_P := t_S$ and $t_{P_E} := t_P + 1$.

The stream time interval of the objects on the right is also manipulated with a window operator. Following the list above, this can be either a partitioned element window of the size 1 or a time window with the size of one time instance.

Following these preparations, the join operator can join the two streams. The join operator considers the stream time intervals of the stream elements so that the trigger behavior can be achieved. The stream now has the necessary schema to calculate the distance between two moving objects: $(tpoint, int, tpoint, int, [t_S, t_{S_E}), [t_P, t_{P_E}))$.

4.6.5 Step 3: Distance Calculation and Radius Predicate

The last two operators are used for the actual radius calculation. The map operator calculates the distance between the two points in one stream element. Here, the lifted distance operation is used because the two points are both of a temporal type. Nevertheless, due to the limitation of the prediction time to one time instance, the result, a *tfloat*, is only valid one time instance. The last step is a select operation that filters out the stream elements where the *tpoints* have a distance greater than a given radius, e. g., 500 m.

The last steps could be combined into one predicate, and, for example, calculated in the join operator. The separation in the example query plan makes the query easier to understand. In the translation process from the query definition to an execution plan, the query optimizer in the DSMS can optimize the query anyway due to the underlying formal algebra. Additionally, the result now has the distance of the two objects within the schema of the stream element. This can be used in additional steps to see which of the selected moving objects are closer to the center than others.

4.6.6 Problems

When executing this query, some problems will occur, especially in the second step. With the interval approach, element windows have a blocking behavior. Before they can write an output, they have to know the end-timestamp [Krä07]. In a time window, the end timestamp t_{S_E} can be calculated. In an element window, the window operator needs to wait for the following element, because the start timestamp from the following element defines the end-timestamp of the current element. The result in this case is that the query cannot use the newest data, but only the last but one latest location.

The partitioning of the element window is another problem. To ensure the order of the start timestamp of the stream time, the element window needs to know the current elements of all groups. If one group stops to send location updates, i. e., one moving object does not send location updates anymore, the window would block, too. This can be seen in the example in Figure 4.14.

The example shows the input and output stream using an element window of size one partitioned by the id of the element (in this case there are elements with the id “a” and “b”). The elements are “streaming” in from top to bottom. When the first stream element arrives, no output can be generated because the end timestamp of the element is not known yet. The next element is from another partition (“b”) and therefore cannot trigger an output for partition “a”. The third element is again from partition “a” and can now trigger the output for the first element as the end timestamp t_{S_E} is now known (the solid line indicates the tuple which is written to the output and the dashed line indicates the tuple that triggered the output). The next element is again from partition “a” and could trigger the next output *(a, 16, 20)*. Even though everything in partition “a” is known to write that output element, it cannot be written as the order of the stream

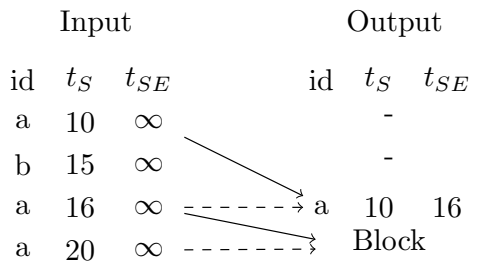


Figure 4.14: Blocking partitioned element window of size one

could be violated. If *(a, 16, 20)* would be written and the next stream element would be from partition "b", the start timestamp would be smaller than the start timestamp from the previous element (16 would be written before 15). As this is not allowed due to the stream properties (the stream needs to be ordered by start timestamp), the element window operator cannot write the next output and will block until an element from parition "b" will arrive. The same is true for the aggregate operator, which also blocks if one group in the partition stops sending data [Krä07].

In a real-world scenario, such a situation can occur. Imagine a vessel moving out of the range of an AIS antenna. No data from that vessel would be received anymore and the query would block.

In addition to these blocking problems, the behavior of the windows cannot be configured for other temporal aspects which are possibly required. For example, it could be useful to limit the moving objects to those which sent a location update within the last hour. This can be useful, for example, in a situation where many objects are in a certain area for a short time and then leave the area. The objects that left the area will stop sending data. To avoid that many very old objects are still considered for the query processing, a window could help to limit the results to the elements that recently sent data.

For the aggregation, the limitation is achieved with the first window in the query plan. But due to the element window, the join also considers stream elements which are older—it is only limited to the most recent stream element from each moving object (partition). The exact formulation would be "From each moving object use the most recent stream element, but only elements which are not older than one hour.".

The conceptual structure here is a good example to illustrate the general handling of queries with multiple moving objects and the problems that can occur with a straightforward solution. Some of these problems can be handled using an alternative query approach.

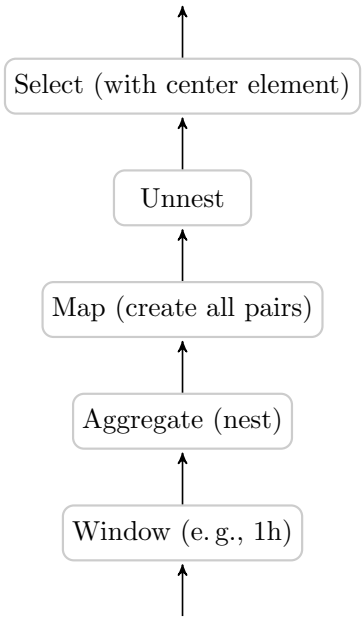


Figure 4.15: Approach with a nest-aggregation

4.6.7 Resolve the Blocking Behavior (at a Cost)

The critical part of the query that brings together the non-center objects and the center objects can be designed differently to resolve the blocking behavior. Instead of using a join operation, an aggregation operation can be used to create pairs of nested stream elements (a stream element containing multiple stream elements). A query using this approach is depicted in Figure 4.15.

The aggregation operation here uses two ideas to solve this problem: (1) it nests the stream element so that a stream element can contain other stream elements. Instead of using a merge function (as, for example, in a join), which creates a new stream element with the fields of the incoming elements, in this case the new stream element is created by putting all elements into it. The second (2) idea is to give the aggregation operation a key for which only one stream element is kept in memory. The key in this example would be the id of a moving object. For each moving object, only one element would be kept in memory, i. e., the latest location update. The nest operation then uses the last valid element (valid in the sense of the window just before the aggregate) of each moving object to create a new stream element. This new element contains all location updates as a list, e. g., *((id1, loc1, t_s, t_{S_E}),(id2, loc2, t_s, t_{S_E}),(id3, loc3, t_s, t_{S_E}),...)*.

In the next step, all possible pairs are created, resulting in a list with a number of lists, each list containing two stream elements. These pairs are unnested so that each stream element now contains a list with exactly two entries (while each entry again is a whole

stream element). The select operation filters out all lists that do not contain a center element. Now, the distance between the two elements in each stream element can be calculated or other spatial operations can be done.

The upside of this approach is that it does not contain any blocking elements. At each newly incoming location update, the new result, for example for a radius query, can be calculated. This is because the element window is replaced by the option for the aggregation to only use the latest element for each moving object.

Nevertheless, this upside comes at a cost. Creating all possible pairs of elements can be computationally too heavy. For each incoming stream element, $n \cdot n$ (with n being the number of different moving objects) pairs are created, which are afterwards filtered again to only those pairs containing a center element. For huge numbers of moving objects, the number of pairs could be too big in a streaming scenario. In comparison, the join approach creates n elements for each incoming center element (joining this element with all latest elements from the other objects) and one element (if the there is only one center element) for each incoming non-center element (joining this element with the center element).

Even though this approach has its downsides, the idea to use an integrated element window within the aggregation operation can be used to improve the first idea of the query with a join operation.

4.7 Non-Blocking Queries with Multiple Objects

The query plan in Section 4.6.2 was straightforward and allows correct results, but at the cost of blocking behavior and the inability to reduce the stream to a time window. Depending on the scenario, these sacrifices cannot be made. The query results are needed immediately—that is one reason why one would use a streaming solution and not a database in the first place. Therefore, a solution is needed that avoids these drawbacks.

The problematic operators in the query plan in Figure 4.13 are the blocking element windows and the join operator that cannot use the information from the time window that reduces the stream to the last one hour or so. This is the central part of the query: it decides how to combine the information from multiple moving objects so that the spatial operations from the moving object algebra can be processed.

To tackle these problems, an extension for the join operator based on the ideas in Section 4.6.7 is proposed. First, the general solution is presented. Then, some special properties of the radius query for moving objects are exploited to improve the general approach for this scenario.

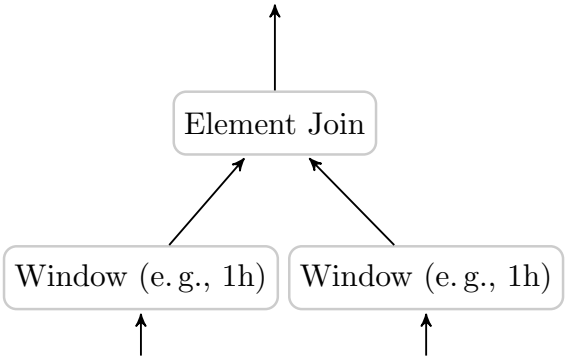


Figure 4.16: Alternative join approach

4.7.1 Element-Join

Figure 4.16 depicts a query with only a join and two separate windows as the inputs. The two incoming streams at the two input ports of the join are independent from each other. One can be, in a temporal sense, faster than the other. The windows are used to reduce the data stream to elements not older than the given time, in this case one hour.

Comparing the queries in Figure 4.13 and Figure 4.16, a main difference is that the left and right window before the join are no element or chronon windows. These windows have been used to control the triggering behavior but wiped the information about the previous window which reduced the stream to a certain time span. This now has to be done in the join operator. Additional to the standard join, the join has to imitate an element window at one or two of the input ports without having a blocking behavior or ignoring the given windows.

For this purpose, the element join is introduced, which solves both problems: the blocking behavior and the limited temporal control. It does this by integrating a partitioned element window behavior into the join itself. Instead of using the interval approach to create element windows by settings correct time intervals in a way that only one or a given number of elements are valid at the same time, the join operator manages the stream elements so that only the given number of elements is used at the same time. This can be configured for each input port of the join operator:

1. **none** No element window is used, the join operator behaves as a normal join, i. e., only considering the time intervals of the incoming stream elements.
2. **left** An element window is only used at the left input port. The right input port behaves normally.
3. **right** An element window is only used at the right input port. The left input port behaves normally.
4. **both** An element window is used on both input ports.

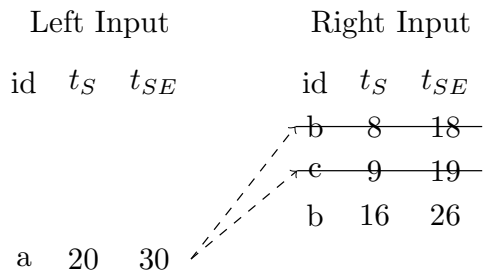


Figure 4.17: Example how the SweepArea is cleaned in a join operator

SweepArea

To understand the way the join operator works with the interval approach, the term SweepArea needs to be introduced. A SweepArea is an abstract data structure that holds a number of stream elements with regard to their stream time interval. That means that a SweepArea stores stream elements, gives access to query them and is able to keep the data structure clean in the sense that old elements which are no longer needed can be removed. A stateful operator typically has one SweepArea per input port, hence a join operator has a left and a right SweepArea [DSTW02].

The SweepArea is cleaned by the time intervals of its elements. When an element with a start timestamp t arrives, all elements with an end timestamp $t_{S_E} \leq t$ can be removed from the SweepArea. In a join, the elements from one port are used to clean the SweepArea from the other port. This is depicted in Figure 4.17. The SweepArea on the right holds three stream elements when the element *(a, 20, 30)* arrives on the left input of the join. Two of the elements on the other side can now be removed from the SweepArea and will not be used for any further join operations. The dashed arrows indicate which elements could be removed from the incoming element on the left.

This behavior now needs to be extended by the element window approach. Doing that, the element window can be seen as a property of the SweepArea. Leaving the partitions aside for a moment, a simple assumption would be that the SweepArea needs to hold exactly n elements for an element window of size n, e. g., one element. If a new element is inserted to the SweepArea, the previous element can be removed.

Unfortunately, this simple approach leads to wrong, or, more precisely, missing results. When removing earlier stream elements, join matches can be missed. This is depicted in Figure 4.18. The first element on the right removes the element on the left (dashed arrow), because its start timestamp is greater than the end timestamp of the left element ($3 < 2$). This is correct. The new, second element on the right removes the previous right element (bend dashed arrow), because we only want an one-element window behavior. Nevertheless, this is wrong, because possible join matches with this element can be missed. The next element on the left, which arrives after that, cannot be joined with the potential join-partner (arrow) *(b,3,7)*, which has already been removed.

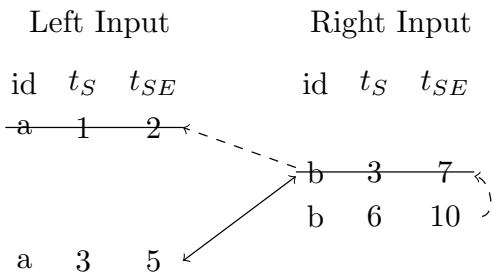


Figure 4.18: Wrong behavior of SweepArea for an element join with size one

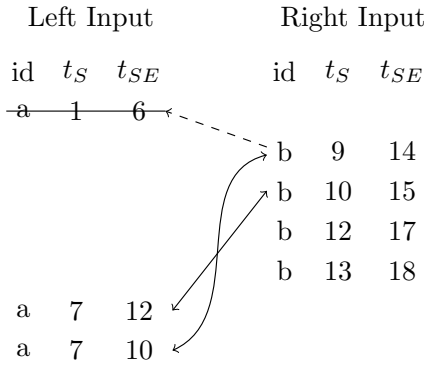
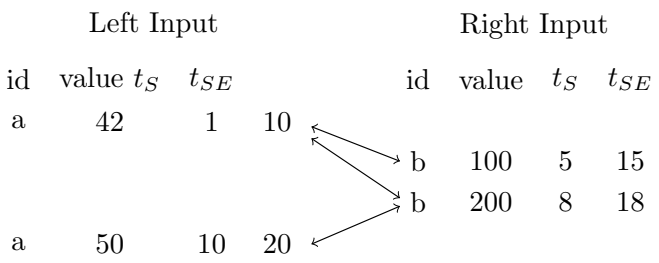


Figure 4.19: Join with SweepArea element size set to one

The SweepArea in the element window needs to be defined a little different. It is not limited to the number *n* of elements, but the join only joins the *n* newest possible matches of the other side. For example, having two possible join partners with an element join configured to the size one, only the newer element is used for the join. Nevertheless, this does not mean that the older elements can be removed at that moment. The example in Figure 4.19 depicts this. The third element on the left side could be joined with two elements: the first and second on the right. Due to the element size set to one, it is only joined with the newer of the two elements. The solid arrows depict the join partners. Nevertheless, the older element on the right cannot be removed at this time, because it is possible that it needs to be joined with later elements on the left. That is the case in this example. The next element cannot be joined with the second element on the right due to its time interval. It needs to be joined with the newest possible element, which is, in this case, the oldest element on the right.

The conclusion from these considerations is that the elements cannot be removed from the SweepAreas based on the number of elements but on the the start timestamp, just like a normal join with normal SweepAreas. The difference is in the matching of the elements to join. In all cases, only the *n* newest possible elements are joined, not all possible elements.



Left Input

id	value	t_S	t_{SE}
a	42	1	10
a	50	10	20

Right Input

id	value	t_S	t_{SE}
b	100	5	15
b	200	8	18

Output

id_left	value_left	id_right	value_right	t_S	t_{SE}
a	42	b	100	5	10
a	42	b	200	8	10
a	50	b	200	10	18

Figure 4.20: Join with SweepArea element size set to one and the output with an end timestamp set

Time Handling

Next to the handling of the elements in the SweepArea, the stream time also needs to be considered for the result elements. Figure 4.20 shows an example of an element join with the element size set to one. For this example input stream, three results are generated. To make it simpler to distinguish the different stream elements, the schema was extended by the "value" field.

In this example, the elements in the output stream use the intersection of the input stream elements as their stream time interval. Even though this reflects the correct join behavior [KS09], in this case it mixes up two semantics. From the perspective of the time interval that is used for the output, the join reflects the semantics of the windows before the join. This way, it is possible to have multiple stream elements with overlapping validities. In Figure 4.20, at the time instances 8 and 9, two output elements are valid. But when having the semantics from the previous windows, some would expect more result elements. In this case, a fourth element could be expected, giving the result of the join of the last element on the left with the first element on the right. But this element is not created due to the limitation of the join to the newest elements. Hence, setting the end timestamps as if there would not be an element limitation in the join is semantically not totally correct, depending on the use case. Nevertheless, in some cases it could be useful to not lose the information of the windows, i. e., merging the time intervals with an intersection just as in the example and by that keeping the end timestamps.

Having an element window inside of the join operator, another solution could be to always have exactly one (or n) valid elements at a time. Then, the end timestamp of an element would be set to the start timestamp of the following element. That would be exactly the behavior of a regular element window and comes with the downside of a blocking behavior.

Another solution is to not decide about the value of the end timestamp. This can be done by setting it to infinity or a value that shows that the end timestamp is not known. It does not lead to any blocking behavior and prevents the time model from having a wrong or mixed semantics.

Algorithm

To include the element window approach into the relational theta-join algorithm for data streams from [KS09], only a few changes are necessary. Algorithm 1 is basically the algorithm from [KS09] with a few modifications.

The size of the "element window" of each of the two input ports is defined in the $elementSize$ variable in Line 4. If all elements have to be used (no element window), this variable is set to -1 (see Line 16). It is important to use the element limitation size from the other input port to get the right number of join partners for a stream element (see Line 19).

In contrast to the original algorithm, all possible join partners ($qualifies$) are sorted so that the newest elements are the first elements of the sorted list (see Line 14). In a real implementation some would avoid the sorting if all elements are used. Nevertheless, for better readability, in this case the elements are sorted in either case. The loop in Line 21 goes through the newest n elements (i. e., the window size) and creates the output stream events. Unlike the algorithm from [KS09], the end timestamp is not derived from the joined stream elements, but set to ∞ (see Lines 25 and 27).

Another approach would be to extend the SweepArea to implement a method that only returns the n newest elements instead of doing the sort in the join operator. Here, the solution within the join operator was chosen to show all modifications in one algorithm.

4.7.2 Optimized Element Join

The approach for the element join above is generalized and ignores some special properties of the original motivation. Taking a look at the original query in Section 4.6.2 again, some special properties of this query can be exploited. First, the join is effectively a self-join. The stream with all location updates is split up and then joined again. That can be used to gain information about the temporal advance later in the query. Second, the predicates of the select operators on the left and right before the join operation are exactly inverse. One select operation selects all elements from the center, the other all

Algorithm 1: Theta Join with Element Limitation

input : physical stream S_{in}, S_{in2}; join predicate θ
output: physical stream S_{out}

```
1  S_out ← ∅;
2  Let SA_i, i ∈ {1,2} be two empty SweepAreas(≤_tE, p^θ_query, p_remove_i);
3  Let Q be an empty min-priority queue with priority ≤_tS;
4  Let elementSize_1, elementSize_2 be the size of the internal element window for
     the left (1) and right (2) side;
5  /* j is this side, k is the other side.                              */
6  j, k ∈ {1,2};
7  foreach s := (e, [t_S, t_E)) ↩ S_in_j do
8      k ← (j mod 2) + 1;
9      SA_k.purgeElements(s, j);
10     SA_j.insert(s);
11     /* Get all possible join partners.                               */
12     Iterator qualifies ← SA_k.query(s, j);
13     /* Sort the possible join partners so that the newest
          elements are first in the list.                               */
14     List sortedQualifies ← createSortedList (qualifies);
15     /* Use only the first (newest) elements from the possible
          join partners or all possible join partners.                  */
16     if elementSize_k = −1 then
17         numberOfElements ← qualifies.size();
18     else
19         numberOfElements ← elementSize_k.size();
20     /* Create the output stream elements.                            */
21     for n ← 0; n < numberOfElements; n ← n + 1 do
22         Element(ê, [t̂_S, t̂_E)) ← sortedQualifies.get(n);
23         if j = 1 then
24             /* Set the end timestamp to infinity.                    */
25             Q.insert((e ∘ ê, [max(t_S, t̂_S), ∞)));
26         else
27             Q.insert((ê ∘ e, [max(t_S, t̂_S), ∞)));
28     t_S_j ← t_S;
29     min_tS ← min(t_S1, t_S2);
30     if min_tS ≠⊥ then
31         /* Transfer all elements with a timestamp ≤ min_tS to S_out
              */
32         Transfer (Q, min_tS, S_out);
33 while ¬Q.isEmpty() do
34     Q.extractMin() ↪ S_out
```

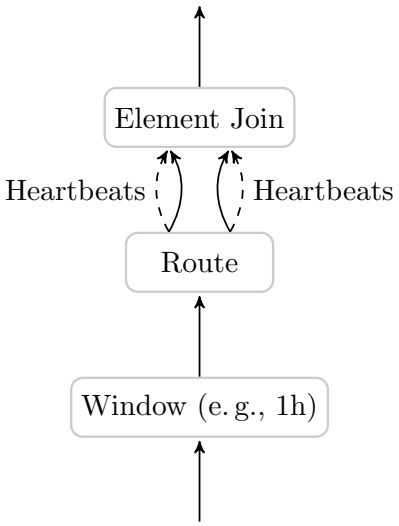


Figure 4.21: Optimized alternative join approach

elements that are not from the center object. This can again be used to gain information about the temporal advance of the data streams.

Considering these properties, an optimized solution for this scenario can be designed. The basic idea is depicted in Figure 4.21. The query only contains the minimal number of operators: a window to reduce the data stream to the last hour or so and by that to ignore moving objects that did not send a location update in that time interval. Next, a route operator is used that replaces the two select operators. The route operator is not a standard operator considering the operator set from [Krä07]. Nevertheless, it simplifies the operator plans by logically combining two inverse select operators. The results are written out on two output ports. Hence, the route does the same as the select operators: separating the center object stream elements and the non-center objects stream elements. This is possible because the predicates of the select operators were exactly inverse. In contrast to the selects, it reduces the number of necessary predicates from two to one and emphasizes that the stream is from one source, i. e., the temporal advance according to their start timestamps is equal. The last operator is the element join introduced in Section 4.7.1 which joins the streams of the two select operators.

The knowledge of the temporal advance of the respective other output port of the route can be used to improve the cleaning process in the SweepAreas. For each stream element that has been filtered out on one side, a heartbeat can be send. Heartbeats are stream elements that do not have payload but only contain information about the temporal advance of a stream. That way, latencies can be reduced because results can be written earlier [SW04]. In this scenario, heartbeats can be used to remove elements from the SweepArea at earlier points in time than in the more general approach.

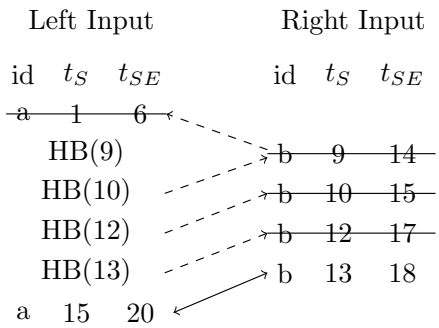


Figure 4.22: Join with SweepArea element size set to one and heartbeats for cleaning

An example which shows a scenario with heartbeats and a SweepArea with an element size set to one is depicted in Figure 4.22. For each regular stream element that flows into the right side, a heartbeat (HB) with the start timestamp from that stream element (in brackets) can be send on the other side. These heartbeats can be used to clean up the other side. In this case, three elements on the right can be deleted due to incoming heartbeats. This can only be done if there is already a newer element in the SweepArea to keep at least one element. It can happen that the heartbeat on the left arrives earlier than the stream element on the right. Take the last heartbeat as an example. If it arrives before the element *(b,13,18)* arrives on the right, the element *(b,12,17)* cannot be removed immediately. In this case, the deletion is done at the moment the newer element on the right arrives in the SweepArea. When a regular stream element on the left arrives, it can be joined with the newest element on the right, as indicated by the arrow.

Being able to remove elements from the SweepArea at an earlier point in time has some advantages. The amount of memory needed to store old stream elements is reduced, which is a critical property for streaming systems. This is due to the in-memory processing with its rather limited resources (cf. 2.2). Additionally, the time to search for the correct join partners is reduced. The SweepArea needs to be queried for the newest n possible join partners. The fewer elements in the SweepArea, the fewer comparisons need to be done. Another advantage is that the join can write out the results earlier as it does not need to wait for previous elements which may be joined later (cf. 4.7.1).

4.7.3 Expiration Function p_{remove} with Early Cleanup

To add the optimized element join to the join algorithm with element limitation (Algorithm 1), the expiration function p_{remove} (see Lines 2 and 9) needs to be changed. The normal remove predicate removes all elements which end timestamp $\hat{t}_E$ is less or equal to the start timestamp t_S of an incoming stream element: remove if $t_S \geq \hat{t}_E$ [Krä07, KS09].

With the optimized element join (see Section 4.7.2), elements can be removed earlier. Let s be an incoming stream element ($s := (e, [t_S, t_E)$)) or an incoming heartbeat ($s :=$

t_S). In either way, the newly incoming stream element contains a timestamp with the current stream time. Let $\hat{s} := (\hat{e}, [\hat{t}_S, \hat{t}_E))$ be a stream element in the SweepArea SA. Let n be the element limitation for the join for the input port for which SA is the SweepArea. Then, the following expiration function can be used:

$$p_{remove}(s, \hat{s}) := \begin{cases} \text{true} & \text{if } (t_S \geq \hat{t}_E) \vee (t_S \geq \hat{t}_S \wedge |SA.\text{getNewerElementsThan}(\hat{s})| \geq n) \\ \text{false} & \text{otherwise} \end{cases}$$

An element can be removed from the SweepArea if the new element has a greater or equal start timestamp compared to the element in the SweepArea ($t_S \geq \hat{t}_E$). This is equal to the normal expiration function. But there is an additional possibility to remove the elements at an earlier point in time. If the new element has a start timestamp greater or equal to the element in the SweepArea and there are at least n elements in the SweepArea that are newer than the element to be removed, the element can be removed from the SweepArea.

The function `getNewerElementsThan(StreamElement s)` needs to be added to the SweepArea. It queries the SweepArea for elements that arrived after the stream element s. This can be done using the start timestamps from the stream elements in the SweepArea. Nevertheless, this is not enough. It is possible that multiple stream elements have the same start timestamp. In that case, the elements still have an order in which they arrived in the SweepArea which needs to be considered in this function. This can, for example, be implemented using a counter (starting at 0 for each new start timestamp) or a list that keeps the order of the elements. This is a difference to the element window behavior in [Krä07] where element windows could cause indeterministic results because of stream elements being indistinguishable when having the same timestamps [Krä07].

4.7.4 Partitioning

The initial motivation for this approach are spatio-temporal queries with multiple moving objects. The idea is to use the newest stream element (i. e., location update) per moving object. Therefore, the element join approach has to consider partitions. The input stream is logically split into partitions which are handled independently from each other. In this case, partitioning allows to join with the newest n elements per partition. To decide which stream element is in which partition, a grouping function is used [Krä07, KS09].

Figure 4.23 uses the example from Figure 4.22 and extends it with partitions. The stream elements are separated to the partitions by their *id*. On the left, only one partition exists (a) and on the right there are stream elements from two partitions: b and c. The cleaning process of the SweepArea is also separated by the partitions. Only if a newer element for the respective partition exists, the older element can be removed. It is just as if there would be a SweepArea for each partition. The matching for the join is done

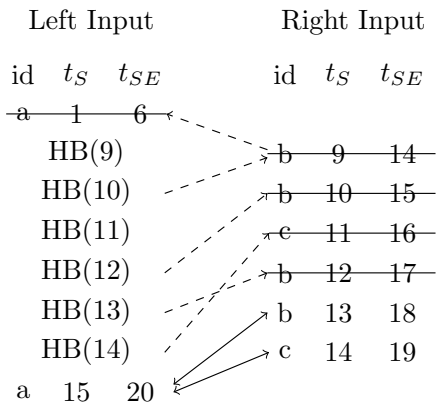


Figure 4.23: Join with SweepArea element size set to one, partitioned by the id and with heartbeats for early cleaning

for each partition as well: for the new element *(a,15,20)* on the left, two join partners are found, one for each partition.

Algorithm 1 can be extended by the partitioning approach, which is done with Algorithm 2. The grouping part of the algorithm is based on the algorithms for grouped aggregation and the grouped element window from [Krä07].

Instead of using one SweepArea for each input port, a map of SweepAreas is used (see Lines 2 and 3) with one SweepArea for each group. The group is determined with the grouping functions f_{group_1} and f_{group_2}. Two functions are needed because each input port can have its own grouping function. First, the correct SweepArea for the group of the new element s is determined. If it does not exist yet, it is created (see Line 16).

The join operation needs to be done for each group individually. The loop in Line 21 loops through all SweepAreas of the other side of the incoming element. For each SweepArea the same procedure as in Algorithm 1 is done. In addition, the algorithm needs to remember the oldest start timestamp (the smallest) over all groups from each side to decide which elements can be written to the output stream and which need to stay in the queue Q (see Lines 37 and 38).

4.8 Filter and Refine

The initial goal of the join queries and extensions described above has been the ability to combine moving objects in order to calculate their distance. The distance in turn can be used to calculate a radius query. The described approaches take into account the temporal aspects of the streams but leave out the spatial properties of the data.

Algorithm 2: Partitioned Theta Join with Element Limitation

input : physical stream S_{in}, S_{in2}; grouping functions f_{group_1}, f_{group_2}, join predicate θ
output: physical stream S_{out}

```
1  S_out ← ∅;
2  Let groups_1 be an empty map with entries < groupID, sweepArea >;
3  Let groups_2 be an empty map with entries < groupID, sweepArea >;
4  Let Q be an empty min-priority queue with priority ≤_{t_S};
5  Let elementSize_1, elementSize_2 be the size of the internal element window for
     the left (1) and right (2) side;
6  /* j is this side, k is the other side.                                  */
7  j, k ∈ {1, 2};
8  min_{t_S} ∈ T ∪ {⊥};
9  foreach s := (e, [t_S, t_E)) ↩ S_{in_j} do
10     k ← (j mod 2) + 1;
11     /* Insert element in SweepArea on this side (j).                     */
12     GroupIdentifier groupID ← f_{group_j}(e);
13     SweepArea SA;
14     if groups_j.containsKey(groupID) then SA ← groups_j.get(groupID);
15     else
16         SA ← new SweepArea(≤_{t_S}, p^θ_{query}, p_{remove_j});
17         groups_j.put(groupID, SA);
18     min_{t_S} ←⊥;
19     SA.insert(s);
20     /* Do join algorithm for all SweepAreas of the other side
           (k)                                                             */
21     foreach SweepArea SA_other in groups_k do
22         SA_other.purgeElements(s, j);
23         /* Get all possible join partners.                             */
24         Iterator qualifies ← SA_other.query(s, j);
25         /* Sort the possible join partners so that the newest
               elements are first in the list.                             */
26         List sortedQualifies ← createSortedList (qualifies);
27         /* Use only the first (newest) elements from the
               possible join partners or all possible join
               partners.                                                   */
28         if elementSize_k = −1 then numberOfElements ← qualifies.size();
29         else numberOfElements ← elementSize_k.size();
30         /* Create the output stream elements.                          */
31         for n ← 0; n < numberOfElements; n ← n + 1 do
32             Element(ê, [t̂_S, t̂_E)) ← sortedQualifies.get(n);
33             /* Set the end timestamp to infinity.                      */
34             if j = 1 then Q.insert((e ∘ ê, [max(t_S, t̂_S), ∞)));
35             else Q.insert((ê ∘ e, [max(t_S, t̂_S), ∞)));
36         /* Save the smallest timestamp of this side over all
               groups.                                                     */
37         if (t_{S_j} =⊥) ∨ (t_S < t_{S_j}) then t_{S_j} ← t_S;
38     min_{t_S} ← min(t_{S_1}, t_{S_2});
39     /* Transfer all elements with a timestamp ≤ min_{t_S} to S_out     */
40     if min_{t_S} ≠⊥ then Transfer (Q, min_{t_S}, S_out);
41 while ¬Q.isEmpty() do Q.extractMin() ↪ S_out;
```

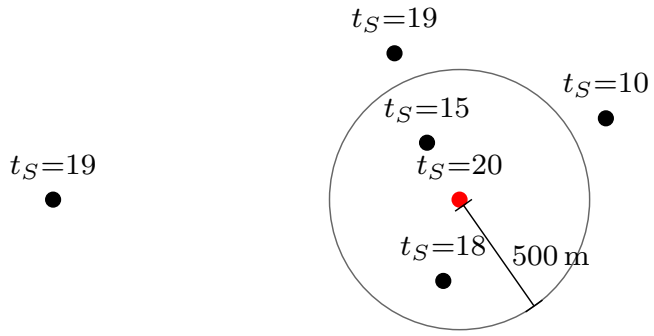


Figure 4.24: Center moving object at $t_S = 20$ with surrounding moving objects and an object far away from the center

In order to improve the efficiency of the query, the number of necessary distance calculations to get a result for the radius query can be reduced [BG19]. Doing this, the accuracy of the result should not be unnecessarily decreased. Depending on the query and spatial distribution of the data, both of these conflicting goals can be met.

Consider the example in Figure 4.24. At $t_S = 20$, a new location update from the center element arrives and is joined with the latest location updates from all other known moving objects. Then, the distance between the center and all these other elements is calculated. To do that, the other elements are predicted to the time instance 20 (by setting their prediction time to $t_P = [20, 21)$). Then, the distance is calculated (e. g., the euclidean distance or the geodesic distance [Kar13]).

Now consider the element on the left. The last known location is only one time instance old and the location is far away from the queried area. Assuming that the elements can only move a short distance in that time (e. g., 50 meters), it is unlikely to impossible that this element will be in the queried region at time 20. The prediction as well as the distance calculation between the two elements is unnecessary as it seems to be clear that this element will not be in the result. Nevertheless, to decide if the object is too far away, the distance needs to be known, resulting in a calculation that was aimed to be omitted.

4.8.1 Filtering in a Streaming Scenario

In spatial databases, a similar problem occurs. The results of a spatial query have to be calculated with as few costly spatial operations as possible. A common approach is to use filter and refine steps (cf. 2.4.5). The candidate estimation in the filtering steps can be done using spatial indexes. These allow quick approximations to filter out elements that are not in the result set (cf. 2.4.5).

Such an index needs to be created and kept up to date with the current data. In a streaming environment this is especially demanding as the data is changing continuously.

Additionally, the temporal dimensions need to be considered. The spatial information is valid at certain time intervals, which are represented by the stream time and prediction time intervals. The spatial index cannot simply assume that all elements are valid at the same time. Instead, it needs to be able to represent the spatial state at certain points in time. Doing that, the size of an index and with that its memory usage needs to be considered. Because in data stream processing most data is kept in memory, the index size should be small enough to fit into this limited resource.

Another distinction to filtering approaches in databases is that the database can reduce the number of data elements that are involved in a query by its first filtering step. A spatial index can be used to reduce the number of elements that need to be read from disk. In a streaming scenario, all elements flow into the query, independent of the fact if they are filtered out later or not. This inverts the function of at least the first filter step: it does not tell *which* elements are potential results (this is done in a database), but for a certain element it tells *if* it is a potential result.

Due to the demanding requirements to a spatial index in a streaming scenario, solutions without an index for certain queries can reduce the overhead of updating an index but keep the general approach of filtering out elements at an early stage.

4.8.2 Filtering for a Radius Query

Taking another look at the radius query from Section 4.6.2 as an example for a query that involves a spatial predicate with multiple moving objects, a feasible placement of a filter would be at or directly after the join operation. That is because this is the earliest point in the query where both spatial objects are known. The filter step can be applied in a temporal or non-temporal manner, i. e., using prediction or not using prediction. Calculating the filter step in a non-temporal manner typically should reduce the costs of the filter step and be the preferred approach. Figure 4.25 depicts a query with one filter step right after the join. Instead of one, multiple filter steps would be possible. The prediction time would be set right after the filtering step to do the refinement step in a temporal manner.

The important part of the filtering step, which is represented by a select operator from the DSMS, is the filter predicate. As drafted in Figure 4.24, it needs to consider both the temporal as well as the spatial distance between the object in the center and the other object. The basic idea is to estimate the maximum distance the two objects could have traveled (towards each other) from their respective known time to the time where the query predicts them to and subsequently to calculate if the “other” object could reach the queried area around the “center” object.

For the filter step, three approaches are presented and discussed in the following: the approximate distance calculation, the single rectangle filter and the multi rectangle filter.

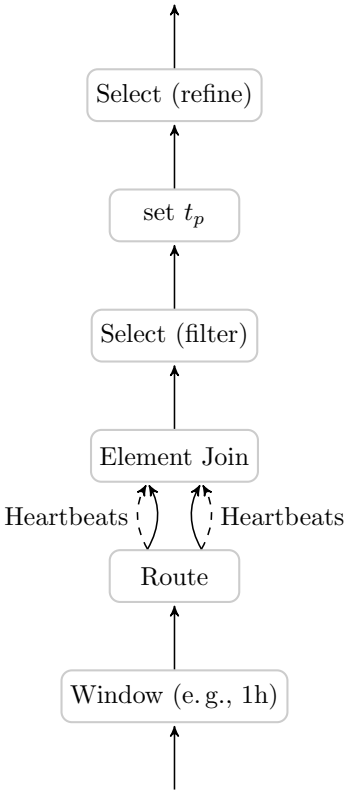


Figure 4.25: Join approach with filter

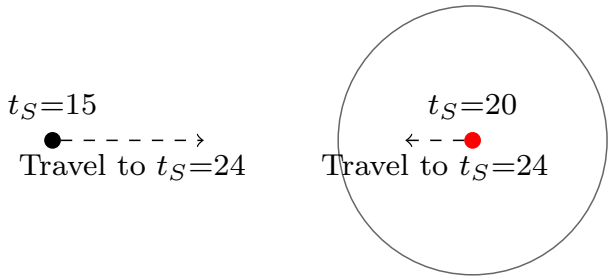


Figure 4.26: Estimation of possible travel distances when predicting to $t_S = 24$

4.8.2.1 Approximate Distance

The goal of the filter step is to remove all objects from the stream which are far away from the center object and to keep those in the stream which are close to it or could be close at the time of prediction. This approach is a straight-forward implementation from the concept explained above. The distance between the center object and the other object is calculated and compared to the radius of the query and the possibly traveled distance of both objects. When calculating the possibly traveled distance of the moving objects, the time span between the point in time of the last known location and the time to which the objects are predicted has to be used.

Estimate Traveled Distance

Having spatio-temporal data, it is not enough to estimate if the last known location of each of the two involved objects are roughly close to each other, but if they could be in the future to which they possibly are predicted. Figure 4.26 depicts a situation where the possibly traveled distance needs to be considered. Both the locations from the other element and the center element are from a point in time which is not equal to the prediction time. Hence, this query asks for a possible future situation. To estimate if the element on the right could possibly be within the radius, is has to be calculated how far the objects can, at maximum, travel. The prediction function is not used as it could be computationally too complex for a cheap filter step. To make this calculation, a maximum possible velocity for the moving objects needs to be estimated.

The calculated traveled distance can be used to estimate how far the other object can be away from the center object (using the measured, not the predicted locations) and still could reach the queried area. As can be seen in the following formula, the maximum distance between the two objects is the sum of the maximum distance each object can move in that time and the radius from the radius query.

$$\begin{aligned} maxDistance = tempDistance_{other} * v_{max_{other}} \\ + tempDistance_{center} * v_{max_{center}} \\ + radius \end{aligned}$$

The temporal distance *tempDistance* between the time when the location has been measured and the time of prediction can be set to $max(|t_S - t_{P_E}|, |t_S - t_P|)$. The maximum of the difference to the start or end timestamp of the prediction time interval is used because the prediction can both be into the future and the past and the maximum time an object could travel is needed. If the prediction time has multiple time intervals (cf. 3.2), the maximum temporal difference over all prediction time intervals needs to be used. Alternatively, each prediction time instance could be handled independently. Nevertheless, this would increase the computational cost without gaining a considerable advantage.

Filter Predicate

Having the maximum possibly traveled distance from both moving objects, the predicate for the select operator that performs the filtering step can be defined as follows: $distance(loc_{center}, loc_{other}) < maxDistance$. The distance between the non-temporal last known location of the two moving objects is calculated and compared to the maximum distance that these objects could travel.

Discussion

An advantage of this approach is that it is simple to implement and straight-forward. It decides for each moving object individually if it could be in the result set or not and thus can be comparably accurate. Additionally, the maximum speed v_{max} of the moving objects can be individually defined, e. g., within the stream element. On the downside, it uses a distance calculation between two objects, which is a comparatively expensive operation. The idea of the filter step is to reduce those operations, not to add more of them. Nevertheless, in this scenario, the downside is less critical if the prediction time interval is large. Because the filter step is completely non-temporal, it is only calculated once per stream element. The refine step is calculated once for each point in time in the prediction time interval. For example, having a prediction time interval with 30 points in time in it (e. g., 30 minutes with a one minute granularity), one distance calculation in the filter step can avoid 30 distance calculations in the refinement step.

4.8.2.2 Single Rectangle

The previous concept for a filter step is straight-forward but uses a distance calculation for every combination of center elements and other elements. Avoiding a potentially expensive geodesic distance calculation could reduce the costs of the filtering step. The

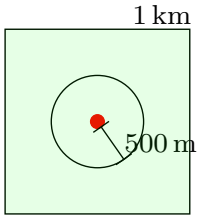


Figure 4.27: Rectangular box around the center element

idea of this approach is to create a bounding box around the center element once for every location update of the center element and then reusing this bounding box for every joined other element. Instead of calculating the possibly traveled distance and calculating the distance between the elements, the only thing that needs to be checked is whether the other element is within the rectangle or not.

This approach is depicted in Figure 4.27. The bounding box has the center moving object in the middle and adds an extra margin around the queried radius, which is depicted as a circle. The size of the margin is critical for the filtering step. A bigger bounding box reduces the probability that some possible results are filtered out but increases the number of possibly irrelevant elements that pass the filtering step. A smaller bounding box reduces the number of elements that needs to be calculated in the more expensive refinement step but increases the risk to loose correct results.

Discussion

An advantage of this approach is its simple implementation and its possibly high efficiency. It is not necessary to calculate possible travel distances nor does the distance between the center object and the other object needs to be calculated. The disadvantage of this approach is that the decision which elements to filter out and which to keep is less fine granular and not individual for each moving object. The margin around the radius is fixed and the same for all elements. While the previous approach with the approximated distances needs to decide the maximum possible velocity of the moving objects, the user has to decide the necessary margin in this approach.

4.8.2.3 Multi Rectangle

While the previous approaches are either very individual but possibly expensive or do not distinguish between the elements but are rather cheap, this approach tries to combine both approaches. To do this, multiple bounding boxes are used and the correct bounding box is chosen depending on the individual possible traveled distance of the moving objects. The approach can be described in three steps:

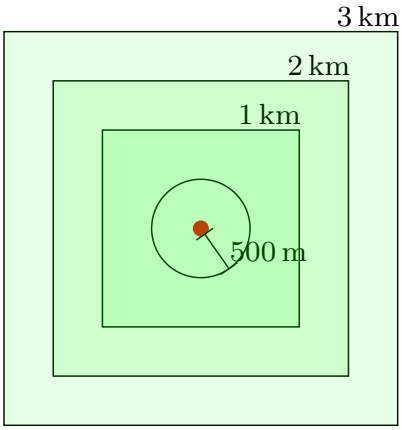


Figure 4.28: Different boxes for different possible travel distances

Step 1: Create Bounding Boxes

Figure 4.28 depicts a center object with the radius from the radius query and three bounding boxes with different minimum distances to the center. These boxes are later used depending on the temporal distance: if the objects could have traveled far, a bigger box is used than if they could only have traveled a short distance. If the objects are too far away to be in any of the boxes, they are not considered. As the prediction has a limited accuracy, using objects which are very far away are very vague results anyway. The number and size of the bounding boxes can be aligned to the use case, the radius in the refinement step and the prediction quality.

Step 2: Estimate Traveled Distance

This step is identical to the estimation of the traveled distance in the approach with approximate distances (cf. 4.8.2.1). In short, for each moving object, the distance is calculated that this object could travel in the time between the last known location and the prediction time.

Step 3: Choose Correct Bounding Box

Based on the approximate distance from Step 2, the correct bounding box is chosen. In the example from Figure 4.28, if the distance is 0.9 km, the first box would be chosen. If the other object is within that box, it is not filtered out but send to the next step, which could be another filter or the refinement step.

Discussion

All the described steps can be applied with standard operations in a DSMS such as the select and map operations with spatial and mathematical expressions. The general

structure of the queries does not need to be changed, which makes the integration of a filter step into a query feasible. The seamless integration into the processing model of a DSMS comes at the cost that it does not allow the calculation of early results (r_1, r_2, ...). If the predicate would be sure that a certain pair of elements is part of the result, it is processed by the subsequent operations anyway.

This approach connects the advantages of the two previous approaches and thus allows a more fine-grained selection of relevant elements than with a single bounding box (cf. 4.8.2.2) while avoiding distance calculations (cf. 4.8.2.1). Nevertheless, it adds more complexity to the query than the other two approaches and may be computationally more expensive depending on the use case due to the need to create more than one rectangle and calculating possible traveled distances.

When deciding for a filtering approach, different queries, data streams and requirements need to be considered. Even through this discussion cannot decide which approach is the best for each use case, it shows that the flexible DSMS approach to process moving object data streams allows for multiple possibilities to include a filter and refinement strategy to a query without the need to add new capabilities to the DSMS itself. Other queries than radius queries, such as kNN, may require different approaches. Nevertheless, queries which work with the proximity of objects can basically use these approaches.

4.9 Conceptual Contribution and Differentiation to Related Work

The concept developed in this and the previous chapter builds upon existing work and extends it for the use with moving object data streams. This section recaps the related work that is closest to this thesis and shows the similarities and the differences to point out the contribution of this work.

A main inspiration for the concept with the moving object algebra is the SECONDO spatio-temporal database [GAA+05]. Despite having a very flexible algebra due to the implemented moving object algebra, the system is designed for static data and not for data streams. Hence, a key difference to SECONDO is the integration of temporal attributes into a streaming system, combining stream time intervals with windowing functionality on the one hand and prediction time intervals on the other hand. Additionally, typical spatio-temporal queries such as the radius and kNN query are implemented with standard DSMS operators.

Tile38 is another system that can run queries on moving object data streams. Nevertheless, designed as a database, it has very limited capabilities when it comes to data stream processing. For example, the system lacks a window concept or any other than the spatio-temporal operations for which the system is designed. Hence, the approach is less generic and the use case limited to the spatio-temporal queries. Nevertheless, Tile38 gives good examples for queries on moving objects, such as static and roaming

geofences, which are similar to an intersection with a non-moving region and the radius query.

MobyDick and OCEANUS from [Gal16] do not discuss queries where the locations of multiple moving objects have to be compared, e. g., to find moving objects that are close to another moving object. This kind of query is an important part of this thesis. For example, Section 4.7 explicitly develops a solution for non-blocking streaming queries which involve multiple moving objects. The prediction concept with temporal attributes is especially used for the case that the location of multiple moving objects are not known at the same point in time. These challenges are not the focus of MobyDick and OCEANUS, even though both works give a first impression for the integration of the moving object algebra into a streaming system. Nevertheless, they do not use the interval approach and therefore do not discuss the integration of multiple temporal dimensions, while the discussion about the algebra when combined with the interval approach and windows is an important contribution of this work.

The integration of two time dimensions is inspired by the Odysseus-based StreamDrive system from [Bol11]. The contribution of this work is that the bitemporal approach from [Bol11] is combined with the moving object algebra to use it for spatio-temporal queries. While [Bol11] is focused on driver assistance systems, this work uses a more generic approach and applies it to moving object queries.

4.10 Summary

This chapter takes the results from Chapter 3 about the logical integration and transfers them to the physical integration, which is closer to the implementation of the theoretical concepts. For this purpose, the transformation from a physical stream to a logical stream has been described in Section 4.1. Having that transformation definition, the results of Chapter 3 can be applied to physical streams.

A main concept of this thesis are temporal attributes. They are basically a function which uses the second temporal dimension, the prediction time. How attributes can be made temporal, how the prediction time dimension is affected by operations and more challenges and solutions with temporal attributes have been described in Section 4.2. To work with temporal attributes, some existing operators need to be extended, among them the aggregation, which has been described in Section 4.3.5.

The prediction time is basically an additional metadata for stream elements and therefore needs to be handled when stream elements are joined. How the prediction time is merged, which is especially used in a join operation, is described in Section 4.3.6. The section discusses different possible merging functions for the prediction time intervals. Another metadata that has been used and extended in this thesis is the temporal trust value, which can be used to represent the trustworthiness of a prediction by a temporal attribute. In summary, a temporal trust value is itself a temporal double value which rep-

resents an estimated prediction quality identifier (i. e., trust) for a stream element with one or more temporal attributes.

The processing of queries on data streams from moving objects pose another stream processing challenge which is independent of the prediction time and the spatio-temporal nature of the data. The calculation of queries which need to join data from different moving objects can cause a blocking behavior, which is very unhandy if event-driven near real-time results are desired. Therefore, this thesis introduces a non-blocking join operator for (grouped) element windows. In short, this allows to join the newest locations of moving objects with each other without waiting for the next location by including some window logic into the join operator itself. This is described in Sections 4.6 and 4.7.

Finally, Section 4.8 gives attention to the second part of the research question and discusses possibilities and limitations on how the performance of continuous spatio-temporal queries on moving object data streams can be improved. It starts by looking at the filter and refine concept in traditional spatial databases and explains how parts of this concept can be integrated into the stream processing environment. Three different approaches for the filtering step are presented at the example of a radius query. They reach from simple to more complex and have different properties when it comes to filtering out the correct stream elements. When deciding for a filter approach, the query and the data properties have to be considered.

The physical concept brought the idea of the moving object stream processing closer to the implementation. Therefore, the next chapter looks closer at the details and challenges of the implementation and explains solutions to make the concept work in a DSMS.

5 Architecture and Implementation

The previous chapters have shown a concept to query spatio-temporal moving objects within a DSMS. To show the feasibility of the concept, it is implemented into an existing DSMS. For the system to be extended, some requirements have to be met: (1) it needs to be build upon the interval approach used in this thesis, (2) it needs to provide the standard relational operators such as select and join and (3) it needs to be extensible. Due to these requirements, the Odysseus DSMS framework is chosen as the foundation for the implementation. It implements the time-interval approach and provides the standard operators from the relational algebra. The open source code and the modular approach make Odysseus extensible, so that the concepts of this work can be implemented into the existing system while the already implemented capabilities can be reused.

This chapter describes the implementation of the concept into the DSMS Odysseus. First, the architecture of Odysseus is described in Section 5.1. Then, the implementation of the integration of the prediction time dimension and the temporal attributes from the moving object algebra is explained in Section 5.2. This section contains details about the new metadata type for the prediction times, about expressions on temporal attributes and temporal functions as well as the implementation of the temporal trust value. The implementations of the extensions for the standard non-temporal operators such as map (cf. 5.2.6), select (cf. 5.2.8), aggregation (cf. 5.2.9) and others are an important part of this section as well. The up- and downsides of the decisions are discussed and differences to the concept are pointed out.

The implementation of the element join (cf. 4.7.2) is described in Section 5.3. This is followed by Section 5.4 about spatial operations. The parts of the implementation described above, even though targeted for moving objects, are not limited to spatial operations. Nevertheless, these are important in this thesis. Hence, the implementation parts which are especially about spatial operations can be found in this section. This also includes the section about spatio-temporal filtering, which aims to improve the performance of the queries.

Names of classes, variables, etc. of the source code are written in a `monotype` font.

5.1 Odysseus

Odysseus is an open-source DSMS developed at the University of Oldenburg. It is written in Java and builds upon OSGi, which makes it a very modular software system [BGJ+09, GHN14, JG08]. Due to this modularity, Odysseus can be easily extended by new operations or concepts without changing or breaking existing code.

The processing concept is based upon the interval approach from [KS09]. Figure 5.1 provides a general overview of Odysseus. The core system of Odysseus is depicted in the middle. The whole process from a defined query provided by the user in a query

language to a running and optimized query is handled by the system. Odysseus differentiates logical and physical operators which represent the logical and physical algebra. Hence, the translation takes a logical query definition in a query language and compiles it into an executable physical query. The translation process is executed in steps and can be extended by new rules which modify the physical plan, for example, to change the order of operators or to choose an operator based on the type of the data stream elements (e. g., key-value or relational tuples). These mechanisms have been used to implement new translation steps necessary for the concept of this thesis.

As depicted, a query is represented by an operator graph that is made of operators and connections between those. Every event (i. e., data steam element) consists of payload and metadata. The metadata contains the time interval in which the element is valid and can also contain more data, such as the data rate of the stream. The results of a query can be send to other systems, for example, to store them in an archive, raise alarms or monitor other systems.

An important goal of the architecture of Odysseus is that the DSMS is very modular and extensible. Odysseus defines interfaces that can be used to create modules that extend the core system in many aspects. For example, new operator bundles can be added so that Odysseus has more processing capabilities. The input and output adapters can be extended to communicate with more systems, i. e., sources and sinks. Query languages can be added as well as other functionality such as recovery [BGA17], machine learning [GHN14], driver assistance systems [BAG+12] and so on.

Odysseus supports multiple query languages and has a framework to include new languages as well. Next to the SQL-like Continuous Query Language (CQL), Odysseus supports the Procedural Query Language (PQL). In PQL, the query graph is defined by the user by defining and connecting the included operators. The examples of queries in this chapter are given in PQL.

5.2 Temporal Implementation in the DSMS Odysseus

An important goal of the implementation of the concept was to not break the existing system. The spatio-temporal algebra is an optional addition to the system, but Odysseus has to work without this extension, too. Therefore, the modularity of Odysseus is exploited to add certain functionality as new modules. These are new metadata, the handling of temporal attributes and some extensions to existing operators as well as new functions.

5.2.1 Metadata

The metadata in Odysseus adds meta information to each data stream element. This is mainly the stream time for the interval approach. Additionally, the metadata can be used to attach information about the latency and the data rate as well as the probability

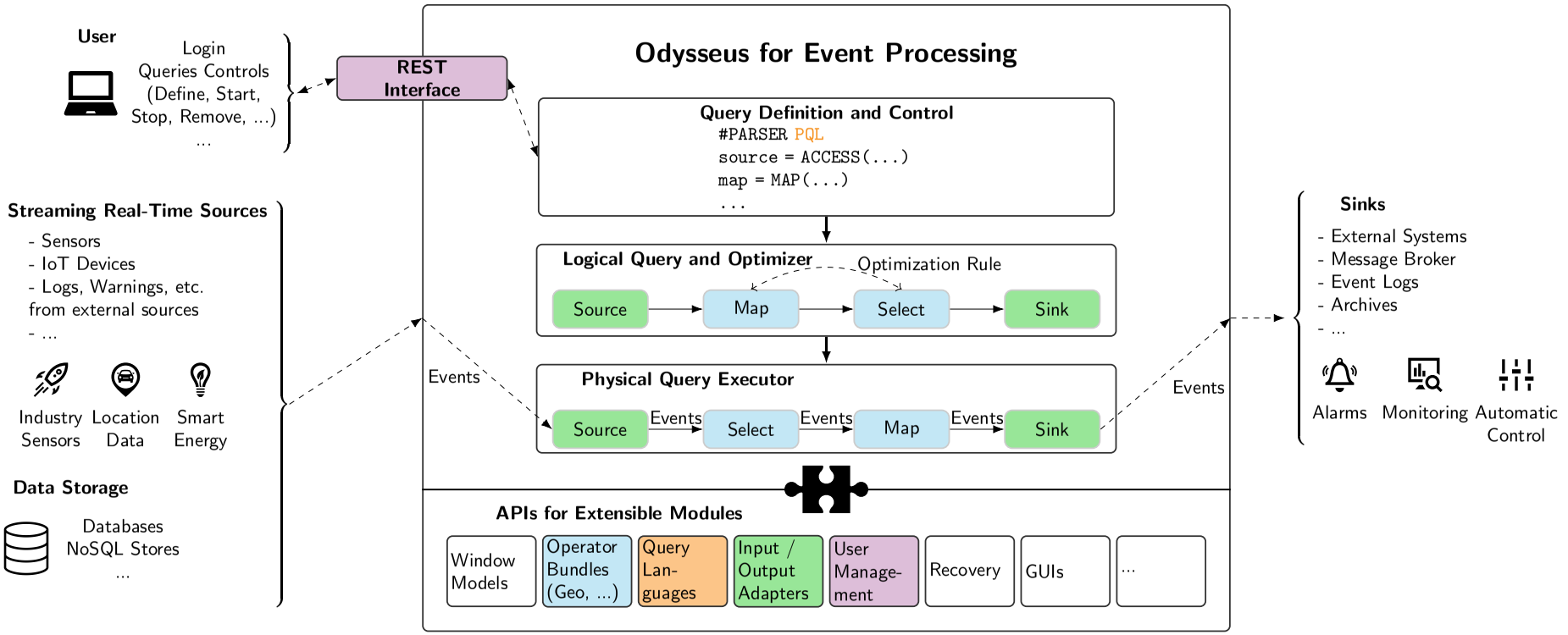


Figure 5.1: Overview of Odysseus

for probabilistic stream processing [Kuk15]. The metadata is extended by the `PredictionTimes` class. It contains a list of `PredictionTime` instances, which define the time intervals to which the temporal types of the stream element have to be predicted. The `PredictionTime` class uses the behavior of the existing stream time implementation from the `TimeInterval` class.

During the definition of a query, the user can define that the prediction time is used. It can be arbitrarily combined with other metadata such as the stream time `TimeInterval`. Listing 5.1 gives an example access operator. Due to the definition of the metadata in Line 11, the `PredictionTimes` metadata is used in this query. Having this metadata, temporal attributes can be used and will be predicted to the given prediction time.

```
1  ACCESS({
2     ...
3     schema=[
4        ['id','Integer'],
5        ['x','Integer'],
6        ['y','Integer'],
7        ['start','StartTimeStamp']
8     ],
9     inputschema=['Integer','Integer','Integer','Integer'],
10    ...
11    metaattribute = ['TimeInterval',  'PredictionTimes'],
12    options=[...]
13 })
```

Listing 5.1: Using the PredictionTimes metadata in an access operator

The temporal operators can manipulate the prediction time according to the definition in the concept. Additionally, the user can manipulate the prediction time manually with the `PredictionTime` operator. The metadata field `PredictionTimes` is initially empty. Therefore, it is necessary to use the `PredictionTime` operator before using any operations on temporal attributes. The prediction time needs to be set to a time interval that fits to the purpose of the query. Typically, the prediction time is aligned to the stream time, i. e., the temporal advance of the data stream.

Listing 5.2 shows an example of the `PredictionTime` operator that sets the prediction time of the stream elements aligned to the current stream time. In this case, the `PredictionTimes` will only include one time interval. The parameters `addToStartValue` and `addToEndValue` are mandatory and define a time which is added to the start timestamp of the stream time to calculate the start and end timestamp of the prediction time. If the prediction time has to be aligned at the end timestamp of the stream timestamp (and not at the start timestamp), this can be set with the parameter `alignAtEnd`. The granularity of the prediction time can differ from the granularity of the stream time

(see 4.3.3). The granularity, i. e., the base time unit, of the prediction can be defined with the parameter `predictionBaseTimeUnit`.

In this example, the start timestamp of the prediction time will be exactly the start timestamp of the stream time. The end timestamp of the prediction time will be 10 seconds in the future. The prediction time is aligned at the start timestamp of the stream time and the granularity is set to seconds.

```
1 /// Set the prediction time
2 PredictionTime({
3   addToStartValue = [0, 'seconds'],
4   addToEndValue = [10, 'seconds'],
5   alignAtEnd = false,
6   predictionBaseTimeUnit = 'seconds'
7 },input)
```

Listing 5.2: Initial manipulation of the prediction time metadata

The following example shows a stream element before being manipulated with the `PredictionTime` operator. The prediction time is not set yet, hence, it is initially empty (the empty list is written as []).

some attribute	stream time	prediction time
42	[10,12)	[]

After flowing through the `PredictionTime` operator defined above in Listing 5.2, the prediction time is set:

some attribute	stream time	prediction time
42	[10,12)	[[10,20)]

All following temporal operations will use the prediction time to predict temporal attributes to this time interval.

5.2.2 Temporal Attributes

Odysseus does not support temporal attributes (cf. 4.2) by default, but allows to define arbitrary attribute types. In contrast to other types, a temporal type in fact represents another, non-temporal type. For example, a temporal integer represents an integer attribute. Therefore, the type system itself is not extended. For the standard operations, e. g., functions in a map operator, a temporal integer is simply an integer. This is a crucial detail in the implementation. When a function sees a temporal integer simply as

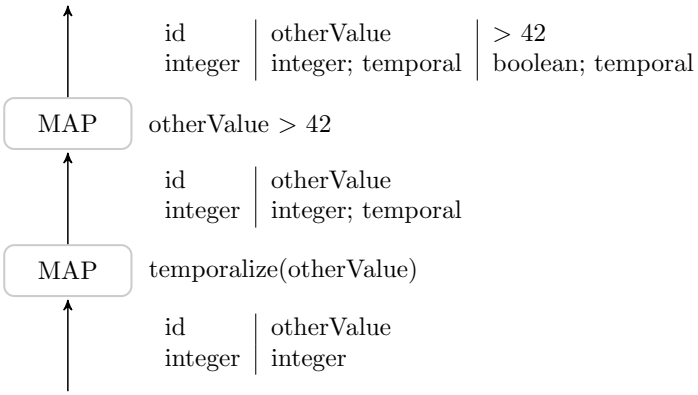


Figure 5.2: Temporal constraint to mark temporal attributes

an integer, the function does not need to be changed. For example, let the "+" function accept only "integer" attributes. If a temporal integer would be a new type of its own, the "+" function would need to be changed to accept this type. Using the non-temporal types in the schema makes the "+" operation simply accept the temporal attribute. This increases the reusability of the existing code and minimizes the changes to the system.

The information that an attribute is temporal is instead added with a so-called constraint, a simple key-value field which is part of the schema for each attribute. It is written during the transformation process, starting with the initial temporalization process and then added down the query graph according to the operations which include temporal attributes. This can be seen in Figure 5.2. At the beginning (bottom), no attribute is temporal. The initial MAP function `temporalize` adds the temporal constraint, i. e., marks this attribute to be a temporal attribute. During the translation process, all other attributes that are (partly) made of this initial temporal attribute, are also marked as being temporal. In this example, the "> 42" attribute is marked as being temporal, because it is created from an expression that includes a temporal attribute ("otherValue").

Translation Process

In Odysseus, the translation process from a logical to a physical query is performed in multiple steps. The steps are defined in the `TransformRuleFlowGroup`, currently consisting of 13 steps. At each step, all available transformation rules for that step are called for the available operators and can, if they are executable at that point of the transformation and for that query, work on the query, or, to be more precise, on one operator of that query.

The temporal constraint is added in the `INIT` phase of the translation process from a logical to a physical query and with that as the second step. The two rules `TSetTemporalConstraintsOnMapAORule` and `TSetTemporalConstraintsOnAggregationAORule` go through the logical schema of a single operator and check for each

expression or aggregation function if temporal attributes are involved. They detect temporal attributes by their temporal constraints, which have been set by a temporalization function or an earlier run of these two rules (the rules can be called multiple times for a translation process, because there can be multiple map or aggregation operators in a query). If an output attribute is temporal, the temporal constraint is set at that attribute in the output schema of that operator.

Having the temporal constraints set in the `INIT` phase of the translation process, the constraints are already set when the logical operators are translated to physical operators in the `TRANSFORMATION` step. This is important to choose the correct temporal operators (e. g., the temporal map operator, see 5.2.6). For example, based on the presence of a temporal constraint, the transformation rule `TTemporalMapAORule` for the temporal map operator is activated, which creates a temporal map operator instead of a non-temporal one.

5.2.3 Representation of Temporal Types

Temporal types are represented by classes that implement the generic `TemporalType` interface. A temporal type can be implemented with a certain function, for example, a linearly moving geometry can be represented using the `TemporalGeometry` class, which in turn uses a temporal function to calculate the geometries (e. g., points) at given points in time. The main method of the interface is `T getValue(PointInTime time)`, which returns a value of the type for a given point in time.

When working with temporal attributes in the query, e. g., with a temporal map operation, it is not feasible or useful to always create a temporal function for the result. For example, for the distance between two moving points, it is not feasible to create a function which represents the distance variations over time precisely. Instead, a simple map from the points in time to the values of the results is used. This is the `GenericTemporalType` which can be used for all types and is typically used for the results of operations.

Listing 5.3 contains an example of the content of a `GenericTemporalType`. Because every single value is stored, instead of having a function to calculate the values, it can happen that more values are stored than necessary. For example, if the `GenericTemporalType` contains values for ten points in time. Then, a select operation reduces the `PredictionTimes` to one point in time. While following operations will only work with that one point in time, the other values are still stored in the object, but will never be used again. To reduce the memory consumption of a query and also to avoid cluttering the view of a `GenericTemporalType` when reading the stream as a user, the unnecessary values can be removed. For this purpose, the `GenericTemporalType` class implements the `trim` method, which removes all stored points in time which are not in the `PredictionTimes` intervals. The `trim` method can be called by the `trimTemporal` function within a map operator.

```
1491005925 = 9451.13
1491005926 = 9435.16
1491005927 = 9439.46
1491005928 = 9421.31
1491005929 = 9404.76
```

Listing 5.3: Example content of an `GenericTemporalType`

PredictionTime Granularity

As described in Section 4.3.3, the prediction time granularity can differ from the granularity of the data stream itself to reduce computational costs. Internally, the points in time that are used to calculate, store and access the values from temporal attributes are always in the time unit of the data stream. For example, if the data stream is in milliseconds and the prediction time is in seconds, when accessing the values for second 1 and 2, the temporal attributes are asked for the values at the points in time 1 000 and 2 000.

Always calculating the timestamp in the stream time eases the process of working with stream elements that have differing prediction time units. As the temporal functions always work with the stream time unit, it is irrelevant, for example, if the prediction time unit changes during the query. This could happen with a `PredictionTime` operator (cf. 5.2.1) which can change the `PredictionBaseTimeUnit` or when merging two `PredictionTimes` metadata fields (cf. 5.2.13).

In cases where a high prediction time granularity is needed, the stream time granularity needs to be increased (e. g., to nanoseconds). A higher granularity for the prediction time than for the stream time is not possible. Nevertheless, increasing the stream time granularity typically does not add expensive side effects (as opposed to the prediction time granularity). For the stream time, simply two timestamps are stored. If they represent milli- or nanoseconds does not change the processing of the stream. Hence, it should not be a problem to increase the stream time granularity if needed.

5.2.4 Temporalization Functions

A non-temporal attribute can be made temporal by using a temporalization function. This can be either an aggregation function or a MAP function, depending on whether it needs a history from a window or not. For moving points there is, for example, the `ToAcceleratingTemporalPoint` aggregation function. It is a simple example function, which uses the speed difference between the first half and second half of the window to calculate the acceleration of the moving object. It is possible to implement more complex algorithms to predict the objects movement, nevertheless, this has not been the focus of this work.

Other examples for temporalization functions are `ToLinearTemporalPoint` and `ToLinearMovingRegion`. These can be directly used in the aggregation operator, as can be seen in Listing 5.4. Here, the attribute `SpatialPoint` is made temporal using the `ToLinearTemporalPoint` function and the result is written to the `temp_SpatialPoint` attribute. The operation is grouped by the `id`, because the trajectories of different moving objects should not be mixed up. This operator should be used with a window before it to reduce the considered and stored stream elements, for example, to the last five minutes.

```
1 /// Temporalize a spatial point
2 temporalize = AGGREGATION({
3   aggregations = [
4     ['function' = 'ToLinearTemporalPoint', 'input_attributes' = '
          SpatialPoint', 'output_attributes' = 'temp_SpatialPoint']
5   ],
6   GROUP_BY = ['id']
7 }, input)
```

Listing 5.4: Temporalization of a spatial point

Another option is the `ToSplineTemporalPoint` aggregation function to create a temporal function using splines. Doing this, not only a straight line, but also a curve can be predicted.

All these temporalization functions create temporal attributes. These are represented by classes that implement the `TemporalAttribute<T>` class. For the mentioned temporalization functions, the temporal functions are implemented in the following classes:

- `AcceleratingMovingPointFunction`
- `LinearMovingPointFunction`
- `SplineMovingPointFunction`
- `LinearMovingRegionFunction`

All of these implement `TemporalAttribute<GeometryWrapper>`. The class `GeometryWrapper` encapsulates the JTS `Geometry` class for Odysseus (cf. 2.4.1).

5.2.5 Temporal Attributes from External Source

The previously described methods to make an attribute temporal are based on the assumption that the future trajectory of a moving object is not known in advance, but can be estimated using the history and current movement of the object (cf. 5.2.4). Nevertheless, in some cases the future trajectory from an object is known with high accuracy, for

example, in cases where the moving object follows the suggested route of a navigation system [SSBK12].

The approach of temporal attributes is generic enough to also use this idea to create a temporal spatial point. As a proof-of-concept, the function `FromTemporalGeoJson` reads a GeoJSON string with additional temporal information to create a temporal point. The trajectory needs to be given as a LineString (cf. 2.4.1), and the timestamp of each point needs to be in an array (in the same order as the points) in the `properties` part of the feature in an array called `times`. Because there is no official specification on how to insert the time domain into GeoJSON, this format follows the handling of times in the TimeDimension add-on for the Leaflet library[1]. Listing 5.5 gives an example of a GeoJSON string with temporal information. This could, for example, be the calculated route of a navigation system with a timestamp at every point or a future trajectory based on a bus schedule.

In this example, the timestamps of the locations are given in an absolute value in UNIX time milliseconds (milliseconds since January 1^st 1970 [Gro]). Conceptually, it would also be possible to use relative times between the locations. Then, the prediction could start at an arbitrary point in time and the relative times would be added up to the start timestamp of the trajectory. This would make it possible to reuse such a trajectory without the need to change the timestamps.

[1] `https://github.com/socib/Leaflet.TimeDimension#ltimedimensionlayergeojson`

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "times":[
                "1534255083000",
                "1534255093000",
                "1534255003000"
            ]
      },
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [
            8.188934326171875,
            53.48722843308561
          ],
          [
            8.206787109375,
            53.55581022359457
          ],
          [
            8.177947998046875,
            53.63161060657857
          ]
        ]
      }
    }
  ]
}
```

Listing 5.5: GeoJSON with additional temporal information

The movement in-between, as well as before and after the known locations could, similar to the approaches without previously known trajectories, be calculated using different interpolation and extrapolation methods. The implementation in the `FromTemporalGeoJson` class creates a `TrajectoryMovingPointFunction` which uses a linear interpolation, i. e., a straight movement from point to point with constant speed. The resulting temporal point can be used just as a temporal point created from a stream of updating locations from a moving object. Currently, the implementation sticks to the definition from Leaflet and always interprets the timestamps as absolute timestamps, hence, relative times between the locations are not supported.

```
1 MAP({
2   expressions = [['FromTemporalGeoJson(geoJSON)','tempPoint']]
3   },input)
```

Listing 5.6: A map operator converting a GeoJSON with timestamps (see Listing 5.5) to a temporal point

The function can be used in a map operator, which will output a temporal point. An example can be seen in Listing 5.6 where the incoming data stream element has an attribute `geoJSON` containing an appropriate GeoJSON string. The temporal point will be in the `tempPoint` attribute of the output stream element. When the trajectory updates, the temporal point can be re-created with a new GeoJSON string flowing into the system. Potentially, other formats or data sources are possible to receive timestamped trajectories for moving objects.

5.2.6 Temporal Map Operator

The map operator applies given functions on attributes of a stream element. The standard relational map operator is replaced by the `TemporalRelationalMapPO` if at least one expression works on a temporal attribute. The temporal map operator extends the standard relational map operator but overrides the `init` method. The difference is that for each expression with at least one temporal attribute, a temporal expression is created.

A `TemporalRelationalExpression` in turn extends the `RelationalExpression` so that for each point in the `PredictionTimes`, the expression is evaluated with the non-temporal values of the attribute. This process is shown in Algorithm 3. The result is always a `GenericTemporalType` (created in Line 1 and filled in Line 5) with a non-temporal value for each point in the prediction time. The non-temporal stream element is created in Line 3 by solving the temporal function for the given point in time `time`. The expression is evaluated with the non-temporal stream element in Line 4 with the evaluate method of the super class `RelationalExpression`. This way, all existing expressions still work with temporal attributes.

The following operators in the query graph consider temporal attributes as well and would also use the temporal version of the operator. Using this approach, the user does not need to manually decide for each operator or expression if it is a temporal or non-temporal one, but the translation process in the DSMS does this automatically.

Again, the temporal operator is used automatically, so that the user can define the operator as if it would be non-temporal, as can be seen in Listing 5.7. Here, the orthodromic distance between two temporal points is calculated. The result in turn is also temporal. At this point in the query, the prediction time should already be set with an PredictionTime operator. If this is not done, the prediction time interval will be empty and hence,

Algorithm 3: Evaluation of a temporal expression

input: streamElement T

```
1 GenericTemporalType result = new GenericTemporalType();
2 foreach PointInTime time in streamElement.getPredictionTimes () do
3     Tuple nonTemporalTuple = createNonTemporalTuple (streamElement,
        time);
4     singleResult = super.evaluate (nonTemporalTuple);
5     result.setValue (time, singleResult);
6 return result;
```

no result will be calculated (which is semantically correct but probably useless to the user). The result of the operation is written to the temporal attribute "tdistance" and is a temporal double value.

```
1 /// Calculate the distance for the prediction time
2 calculateDistance = MAP({
3   expressions = [
4     ['OrthodromicDistance(center_temp_SpatialPoint, temp_SpatialPoint
          )', 'tdistance']
5   ],
6 }, input)
```

Listing 5.7: Map operation on a temporal attribute

Loosing Information

The result of a temporal map operation is always a `GenericTemporalType`, i. e., a map from timestamps to values. Even if the temporal attribute on which the expression is applied is represented by a function. Doing this, information is lost after the expression. For example, the function would be able to calculate values with a higher granularity or at different points in time than defined by the prediction time intervals. The `GenericTemporalType` only includes the values for the timestamps in the prediction time intervals. Nevertheless, the information that is lost is actually not needed later in the query. The granularity is typically not changed within the query and the prediction time intervals should not be manipulated manually after a temporal operation (cf. 4.3.2).

5.2.7 Combining Temporal Expressions

The way the temporal functions and operations have been implemented makes it possible to combine the functions. For example, the result of a "+" operation on two temporal

attributes can be the input of a "≤" operation, which in turn results in a temporal Boolean, which again can be the input for another temporal operation, and so on.

An example is shown in Listing 5.8 and Listing 5.9. Both queries have the same result. "x1" and "x2" are attributes with temporal integers. Therefore, the `TTemporalMapAORule` is activated and translates the relational expression(s) to temporal relational expressions. The query in Listing 5.8 has multiple consecutive expressions, which use the previous result as the new input. The part `x1 + x2` is solved to a temporal integer, which is the input for the part `map1 > 42`. This part is solved to a temporal Boolean result. The non-temporal negation operation "!" can be applied to this temporal Boolean and negates each Boolean value at every point in time in the prediction time interval. Thus, this example shows that temporal attributes can be used in expressions without the need to explicitly declare the operations temporal or use different operations for temporal attributes.

```
map1 = MAP({
  expressions = [['x1 + x2', 'map1']]
}, input)

map2 = MAP({
  expressions = [['map1 > 42', 'map2']]
}, map1)

map3 = MAP({
  expressions = [['!map2', 'map3']]
}, map2)
```

Listing 5.8: Multiple temporal expressions in multiple map operators

This is even possible without splitting the parts of the expression into multiple map operators. The same expression with the same result is shown in Listing 5.9. Here, the intermediate results are not explicitly solved to temporal attributes. Nevertheless, due to the implementation shown in Algorithm 3, the result of the expression is the same. This is possible as long as all expressions need non-temporal values and are therefore actually calculated on non-temporal instances of the temporal attributes. An example of an expression that cannot be combined is given a few sections later in Section 5.2.12.

```
MAP({
  expressions = ['!((x1 + x2) > 42)']
}, input)
```

Listing 5.9: Multiple temporal expressions in one map operator

5.2.8 Temporal Select Operator

The temporal select operator works similarly to the temporal map operator (see 5.2.6). If the predicate of the operator contains at least one temporal attribute, the predicate (which is an expression with a Boolean return value) is changed to a `TemporalRelational-Expression`. The temporal select operator differs from the non-temporal counterpart in the way that it does not expect a Boolean result, but a temporal type with a Boolean return value for every prediction point in time, typically a `GenericTemporalType`.

The temporal select operator takes all values where the expression returns `true` and creates the `PredictionTimes` from it. The new prediction times are a subset (possibly equal) of the prediction times of the incoming stream element. If the expression does not return `true` for any point in time in the prediction times, the stream element is not send to the output, which is just like the behavior of the non-temporal select operator.

Listing 5.10 gives an example on how the temporal select operator can be used in PQL. In this case, the operator is used just after the previously shown temporal map operator in Listing 5.7. The predicate works on a temporal attribute ("tdistance") and manipulates the prediction time intervals in the metadata of the stream elements.

```
1 distanceSelect = SELECT({
2   predicate = 'tdistance < 5000'
3 }, calculateDistance)
```

Listing 5.10: Select operation on a temporal attribute

5.2.9 Temporal Aggregation Functions

Aggregation functions, such as a sum or an average, can also be done over temporal attributes (cf. 4.3.5). In Odysseus, aggregation operations can be executed using the `Aggregation` operator. The operator handles the stream elements and has an interface for aggregation functions. These functions have three important methods: `addNew` to add a new stream element to the function, `removeOutdated` to let the function know which elements need to be removed from the result and `evaluate` to get the current result.

When using non-temporal aggregations over a temporal attribute, a wrapper is needed which splits the temporal stream elements into a set of non-temporal stream elements, which can be used in the aggregation functions. During the translation process, the `TTemporalAggregationAORule` is activated if the attribute of an aggregation function is temporal. In that case, the normal aggregation function is interchanged with a temporal aggregation function which takes the non-temporal function as an argument.

The `TemporalIncrementalAggregationFunction`[2] acts as a wrapper around a non-temporal aggregation function. To do this, it uses a number of copies of an aggregation function, one for each point in the prediction time intervals of the currently valid (in the sense of the stream time) stream elements.

Adding New Stream Elements

The algorithm for adding a new temporal stream element to an aggregation function can be seen in Algorithm 4. When a new stream element is added to the temporal aggregation function via the `addNew(T newElement)` method, where `T` stands for a stream element with a `PredictionTimes` metadata field, the prediction time intervals `PredictionTimes` are read from the metadata of the given stream element (Line 1). Then, for each point in the prediction time, the non-temporal stream element is created by calculating the temporal functions for all temporal attributes (Line 2). For each point in the prediction time an aggregation function exists, i. e., a bare copy of the original aggregation function is created for each point in time. This function is retrieved in Line 3 and filled with the new non-temporal tuple in Line 4. To remove the non-temporal elements from the aggregations later on, it needs to be known which non-temporal elements belong to which temporal element. This is done in Line 5.

Algorithm 4: Adding a new temporal stream element to the non-temporal aggregate functions

input: newElement T

```
1 foreach PointInTime time in newElement.getPredictionTimes () do
2     Tuple nonTemporalTuple = createNonTemporalTuple (newElement,
        time);
3     AggregationFunction agg = getAggregationFunctionForTime (time);
4     agg.addNewTuple (nonTemporalTuple);
5     storeNonTemporalTuple (newElement, nonTemporalTuple);
```

Removing Outdated Stream Elements

When a stream element is outdated, the aggregation operator calls the `removeOutdated` method, which is shown in Algorithm 5. Removing the stream elements from the aggregations does the exact opposite of adding them: for the temporal stream element that needs to be removed, all non-temporal elements are retrieved which belong to this temporal stream element (Line 2). For each point in the prediction time of the temporal stream element, a non-temporal stream element exists in this map. The non-temporal

[2] Most aggregation functions are implemented in an incremental manner to reduce the computational load in contrast to non-incremental algorithms.

element is then removed from the according non-temporal aggregation function for this point in the prediction time (Line 6).

When the prediction times change over time so that some points in time are no longer covered by any valid stream element, the old aggregation functions for these points in time can be removed. This is done in Line 7.

Algorithm 5: Removing an outdated temporal stream element from the non-temporal aggregate functions

input: outdatedElement T

```
1 /* Counterpart of the storeNonTemporalTuple method in
     Algorithm 4                                          */
2 Map<PointInTime, T> nonTemporalElements = getNonTemporalElements
    (outdatedElement);
3 foreach PointInTime time in outdatedElement.getPredictionTimes () do
4     T elementToRemove = nonTemporalElements.get (time);
5     AggregationFunction agg = getAggregationFunctionForTime (time);
6     agg.removeOutdated (elementToRemove);
7 removeOutdatedFunctions ();
```

Evaluate Aggregation Functions

When the aggregation operator needs a new result for a certain point in time, it calls the `evaluate` method of the aggregation functions, which is shown in Algorithm 6. The result will be a `GenericTemporalType` (Line 1) (cf. 5.2.3), i. e., for each point in time in the prediction temporal dimension a result is stored. The `getAllPredictionTimes` method returns the current prediction time intervals, which can be retrieved by merging all the temporal stream elements which are currently within the aggregation function. Then, for each prediction point in time, the correct aggregation function is retrieved (Line 3), calculates the current result (Line 4) and puts the result into the result object (Line 5).

Algorithm 6: Evaluate a temporal aggregation function

input: streamTime PointInTime

```
1 GenericTemporalType result = new GenericTemporalType();
2 foreach PointInTime time in getAllPredictionTimes () do
3     AggregationFunction agg = getAggregationFunctionForTime (time);
4     Object result = agg.evaluate (streamTime);
5     result.setValue (time, result);
6 return result;
```

Metadata Merging

To calculate the result of the aggregation, the prediction time intervals need to be known (cf. Line 2). These can be calculated by merging the prediction time intervals of all temporal stream elements currently in the temporal aggregation function. Section 4.3.5 describes that a union merge function as well as an intersection merge function would be possible. In either case, the calculation of the merge function within the `TemporalIncrementalAggregationFunction` class does not affect the metadata of the resulting stream element, it is only used to calculate the results for the correct points in time. Hence, it would not change the result when doing a union merge within the `TemporalIncrementalAggregationFunction` but an intersection merge for the `PredictionTimes` in the resulting stream element, as the result of the union always includes the result of the intersection. Nevertheless, to prevent unnecessary computations, the merge function should be the same.

5.2.10 Temporal Join Operator

Similar to the (temporal) select operator (cf. 5.2.8), the (temporal) join operator works with predicates, which can be temporal, i. e., contain temporal attributes. The predicates are evaluated by the SweepAreas that the join operator `JoinTIPO` uses to manage the stream elements. As the whole stream element management and the predicate evaluation happens in the SweepAreas, the join operator itself does not need to be changed or extended when working with temporal predicates. Instead, the predicate needs to be exchanged to be a temporal predicate and the SweepAreas need to be able to work with temporal predicates.

The `TTemporalJoinAORule` overrides the standard transformation rule for the `JoinTIPO` and is activated if the predicate contains at least one temporal attribute. In that case, the join predicate is transformed to a temporal predicate in the `setJoinPredicate` method, similar to the temporal select transformation rule (cf. 5.2.8).

For the SweepAreas, the `TemporalJoinTISweepArea` implementation is used. It extends the standard `JoinTISweepArea` and changes the way the evaluation of the predicate is done so that it can handle the temporal Boolean result of the temporal predicate. The method `doTemporalEvaluation` evaluates the predicate, constructs the correct time intervals (just like the temporal select) and creates an output stream element with the correct prediction time intervals. This is done by only using the points in time in the prediction time interval where the temporal predicate returned true.

5.2.11 Temporal Unnest Operator

A nested stream element is a stream element within an attribute of a stream element. This attribute can be temporal, too. When unnesting a stream element, the operator takes

the nested stream elements from the attribute and creates a new stream element for each nested one. To unnest a temporal attribute, the `TemporalRelationalUnnestPO` creates a non-temporal stream element for each point in time in the prediction time intervals and then does the normal, non-temporal unnest operation.

If, for example, a stream element with a temporal nested attribute has three nested stream elements in it and its prediction time interval contains two points in time, $3 \cdot 2 = 6$ stream elements are created. The operator can be used just as the non-temporal operator, as can be seen in Listing 5.11.

```
1 /// Unnest the tuple
2 unnestTemporal = UNNEST({
3   attribute='temporalNestedAttribute'
4 }, input)
```

Listing 5.11: Unnest operation on a temporal attribute

5.2.12 Direct Temporal Functions

The moving object algebra describes both non-temporal functions, which are lifted to temporal functions, and direct temporal functions (cf. 3.3.1). The non-temporal functions are lifted by the temporal variants of the non-temporal operators as described in Sections 5.2.6 and 5.2.8. The direct temporal functions can work on temporal types without the need for a special operator.

Direct temporal functions, for example the "speed" function, implement the `TemporalFunction` interface. The translation rules and the temporal operators, such as the `TemporalRelationalMapPO`, check if a function is a temporal function. If so, the process of calculating a non-temporal result for each point in the prediction time is omitted and the function works directly on the temporal attribute. The `SpeedFunction` is an example for a temporal function. For each point in the prediction time it calculates the speed of a moving geometry. The result is a temporal real value, just as defined in the moving object algebra: speed: $tpoint \rightarrow treal$[3] [GBE+00].

Listing 5.12 gives an example on how to use a temporal function. It can be used similarly to a non-temporal function in a map operator. It takes a temporal geometry, in this case a temporal point, and the metadata which includes the prediction time. This is necessary due to the architecture of functions in Odysseus, which can only access attributes (including the metadata) that are explicitly given to them. Therefore, the metadata attribute `PredictionTimes` has to be put there as a second attribute of the function. The result is a temporal real value, here written to the output attribute "tspeed".

[3] The original moving object algebra uses the terms "mpoint" and "mreal" for "moving" instead of "temporal", which is just another term for the same idea.

```
speed = MAP({
  expressions = [
    ['speed(temp_SpatialPoint, PredictionTimes)','tspeed']
  ]
}, input)
```

Listing 5.12: Speed function on a temporal point

Other implemented functions that work directly on temporal attributes are `AtMin` and `AtMax`, which calculate the minimum or maximum value(s) of a temporal number (integer, double, ...). The `TrajectoryFunction` consumes a temporal point attribute and converts it to a non-temporal spatial trajectory, i. e., a `LineString`. To be more precise, a list of `LineStrings` is created, because the `PredictionTimes` can potentially have multiple time intervals. The temporal information is lost during this process, but other applications that need non-temporal data types can work with the output. This function is an example of a class that implements the `RemoveTemporalFunction` marker interface. Functions that implement this marker take a temporal attribute and create a non-temporal attribute. Odysseus needs to know this to update the schema and choose the correct way to process the respective attribute.

The direct temporal function can be combined with other functions arbitrarily as long as the respective outputs fit together. For example, a direct temporal function needs a temporal attribute as its input. Internally, allowing such mixed expressions with temporal and non-temporal functions is more complex than expressions that are not mixed. The logic for these expressions is implemented mainly in the `MixedTemporalRelational-Expression` class. Internally, it builds a tree from the expression and solves each part of the expression individually to guarantee that each function has the correct input, i. e., temporal or non-temporal. Mixed expressions can be used in all standard operators that work with expressions, i. e., the map, select and join operator. Listing 5.13 shows an example for a mixed expression. The upper two map operators can be combined into one map operator with the same output.

```
1
2 // Separated
3 calcTraj = MAP({
4   expressions = [
5     ['Trajectory(tempSpatialPoint, PredictionTimes)','traj']
6   ],
7   keepinput = true
8 },predTime)
9
10 MAP({
11   expressions = [['SpatialLength(traj)','len']],
12   keepinput = true
13 },calcTraj)
14
15 // Combined in a mixed expression
16 calcTraj = MAP({
17   expressions = [
18     ['SpatialLength(Trajectory(tempSpatialPoint, PredictionTimes))','
         len']
19   ],
20   keepinput = true
21 },predTime)
```

Listing 5.13: Calculating the traveled distance of a temporal point in a mixed expression

5.2.13 PredictionTimes Metadata Merging

When multiple stream elements are combined, e. g., through a join or aggregation operation, the `PredictionTimes` metadata needs to be merged (cf. 4.3.5 and 4.3.6). The merge functions are implemented in the classes `PredictionTimesIntersectionMetadataMergeFunction` and `PredictionTimesUnionMetadataMergeFunction`. When merging the prediction time intervals, the granularity of the prediction time needs to be considered (cf. 4.3.3 and 5.2.3). In the case that the granularity of the two input metadata elements differ, the coarser time unit is used. That is because the temporal attributes with the less granular time unit possibly do not have the values for the points in time of the more granular time unit. This problem is depicted in Figure 5.3.

The first stream element (“in 1”) has a less granular time unit while the second stream element has a more granular time unit. For the result, the less granular time unit is chosen. The intersection is actually an intersection of two sets with points in time in them. Only those points in time that are in both sets are used.

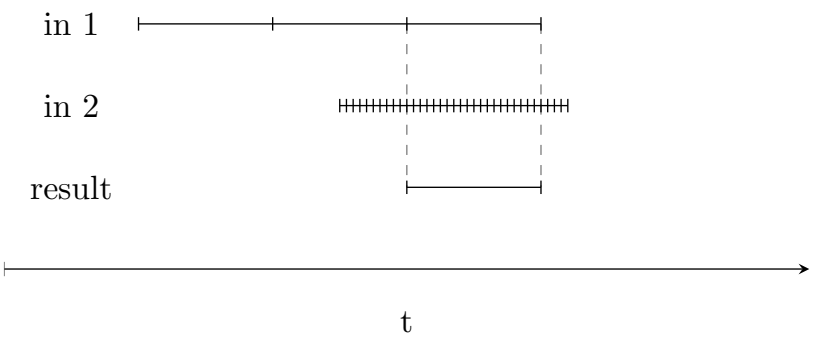


Figure 5.3: Merging two time intervals with different granularities

5.2.14 Temporal Trust Value

The temporal trust (cf. 4.5) value is implemented as a meta attribute in the `TemporalTrust` class. It contains a `GenericTemporalType` which in turn contains values of the type `ITrust`, a meta attribute that represents the trust as a double value. The `TemporalTrustMergeFunction` is used to merge two temporal trust meta attributes and uses the minimal value for each point in time.

The initial trust values come from the temporal attributes themselves. Each temporal type has a trust function which returns a trust for a given point in time. The interface for temporal types, `TemporalType<T>`, has a method `getTrust(PointInTime)` which returns the trust as a double value. This mechanism is independent from the meta attribute `TemporalTrust`, but can and currently is used for the meta attribute.

In the `PredictionTime` operator, which manipulates the `PredictionTimes` intervals, the trust of the temporal attributes in a stream element can be retrieved via the `getTrust(PointInTime)` method. The smallest trust value for each point in the new prediction time is written to the meta attribute `TemporalTrust` if this is used. In other words: for each point in the prediction time, the operator iterates over all attributes in the stream element and picks the lowest trust. This value is used for the temporal trust value for the respective prediction time.

The trust function within a temporal attribute can be any function that implements the `TemporalAttribute<Double>` interface. For example, the proof-of-concept temporalization function `ToLinearTemporalPoint` that creates a `TemporalGeometry` uses a spline function to represent the trust. The trust is very high (1.0) at the point in time when the location of the moving object is known and approaches a low trust (0.0) when the next known location is more than one point in time away. Of course, way more sophisticated estimations of the trust would be possible.

With a temporal expression, a new temporal attribute can be created. For example, using a map operator, two temporal integers can be multiplied. When doing this, the result is again a temporal integer, which has its own temporal function to represent the trust. Again, the minimum trust of all temporal input attributes can be used. This is implemented in the `TemporalRelationalExpression` class.

The `TemporalTrust` metadata can be activated when the `PredictionTimes` metadata is used. The usage can, for example, be defined in an access operator with the `metaAttribute` option. A typical definition for the metadata could be `metaattribute = ['TimeInterval', 'PredictionTimes', 'TemporalTrust']`.

The following table shows an example stream with these metadata attributes:

movement (tpoint)	**stream time**	**prediction times**	**temporal trust**
temporal point 1	[1,11)	[[1,3)]	1 = 1.0, 2=0.5
temporal point 2	[2,12)	[[2,4)]	2 = 1.0, 3=0.8
temporal point 3	[5,15)	[[10,14)]	10 = 0.7, 11=0.6, 12=0.6, 13=0.4

As can be seen, for each point in the prediction time intervals a trust value is in the temporal trust meta attribute. In this example, the trust is higher (closer to 1.0), for times where the prediction time is close to the stream time, which is because in this example the newest known location of the temporal point is at the start of the stream time. From there, the prediction gets less accurate and therefore less trustworthy. The temporal trust value represents this loss in accuracy by reducing the trust value over time.

How the trust values are merged can be seen in the next example. Here, the first stream element is joined with the two other data stream elements. The results of the join can be seen in this table:

movement 1 (tpoint)	**movement 2 (tpoint)**	**stream time**	**prediction times**	**temporal trust**
temporal point 1	temporal point 2	[2,11)	[[2,3)]	2=0.5
temporal point 1	temporal point 3	[5,11)	$\emptyset$	$\emptyset$

The stream time intervals as well as the prediction time intervals are merged by intersecting the stream time intervals of the two input stream elements. For the first result, the prediction times include one point in time. For this point in time, the lowest value of the two input stream elements is used for the trust. For point in time “2”, the values “0.5” and “1.0” are in the input elements, hence, the result is “0.5”. For the second result element, the prediction time intervals are not overlapping, wherefore the prediction time intervals are empty and no trust value is or needs to be calculated.

5.3 Element Join

Using an element window in general causes blocking behavior with the interval approach, wherefore the element join takes the element window logic and puts it into the join operator itself, as explained in Section 4.7.1. Doing this, the blocking behavior can be omitted. To implement this extension, the standard join implementation class `JoinTIPO` ("join time interval physical operator") had to be changed.

5.3.1 Main Parts of the JoinTIPO

The main part of the `JoinTIPO` are the SweepAreas, one for each input data stream. They manage the data stream elements in the windows, remove old data and are queried for the relevant elements for a join operation with a certain stream element.

When a new stream element arrives at the `process_next(T newElement, int port)` method, the SweepArea of the other port is cleaned up by purging all elements which cannot overlap with this all all following stream elements. Next, the SweepArea of the other port is queried for all temporally overlapping elements. These elements are joined with the new element. For each pair of data stream elements, a `dataMerge` function merges the data and a `metadataMerge` function merges the metadata. The resulting new stream element is transferred to the following operators in the operator graph.

5.3.2 The Element Window

After querying the according SweepArea for the elements to join the new stream element with, the number of results needs to be reduced to the newest n elements with n being the size of the element window of that input port. Within the `processOutput` method, this is done with the `reduceToNewestNElements` method. The reduced set of qualified join elements is then joined as in the normal (non-element-window) join.

Nevertheless, the metadata of the output elements need to be changed to infinity to state that they are not part of a window. If they have been part of a window before, the window semantics is now no longer correct, wherefore removing the end timestamp avoids confusion and incorrect behavior of following operators (cf. 4.7.1).

The algorithm from Sections 4.7.2 and 4.7.3 to remove elements that are no longer needed from the SweepAreas as early as possible is implemented in the method `tryEarlyCleanup`. This method is called after each normal purging of all elements which are too old and also in the `processPunctuation` method. This is especially important as it implements the idea that the cleanup can be improved if the join is in fact a self-join with a route operator, which sends a punctuation to all other ports when an element arrives. The `tryEarlyCleanup` itself does simply remove all elements until only the n newest elements are left for that SweepArea.

The size of the element windows in the join operator can be set with the parameters `elementSizePort0` and `elementSizePort1`. It is possible to only use the element window on one input port or on both input ports with different sizes. Listing 5.14 gives an example for a join operator which uses an element window with different sizes for each input port.

```
1 JOIN({
2     elementsizeport0 = 42,
3     elementsizeport1 = 1
4 }, left, right)
```

Listing 5.14: Join operator using element windows for both input ports

5.3.3 Groups

When using an element window, partitions are a helpful concept (cf. 4.7.4). Using partitions, the count for the number of elements in the window is not done over the whole stream, but for each partition or group. That way, it is, for example, possible to keep the newest element of each group. The partitioning can be done using a list of attributes for each input port. For each resulting group, a SweepArea is used to keep the elements of the group. The SweepAreas can be stored in and accessed with a map: `List<Map<Object, ITimeIntervalSweepArea<T>>> groups`. The list contains exactly two maps, one for each input port. The map in turn points from a group identifier (`Object`) to an `ITimeIntervalSweepArea`.

When inserting a new element into a SweepArea, the correct SweepArea for the group needs to be chosen. The method `getSweepArea` returns the correct SweepArea for the given group or creates a new one if no fitting SweepArea exists. If no groups are defined by the user, a standard group for each input port is used, resulting in exactly one group for each of the two input ports. The other operations, such as removing old elements or querying for elements to join with, iterate over all groups of the respective input port.

The user can define the grouping keys with the parameters `group_by_port_0` and `group_by_port_1`. An example is shown in Listing 5.15. For the left port (0), two attributes are used to create the groups, for the right port (1), only one attribute is used. When multiple attributes are used, all combinations of the input data of these attributes form a separate group.

```
JOIN({
    elementsizeport0 = 42,
    elementsizeport1 = 1,
    group_by_port_0 = ['id_left','otherAttribute'],
    group_by_port_1 = ['id_right'],
}, left, right)
```

Listing 5.15: Join operator using element windows and partitioning for both input ports

Using these extensions to the join operator, the scenario from Section 4.6 can be solved without a blocking behavior. The newest location updates from each moving object can be used for the join by using an element size of 1 and the identification numbers of the moving objects as grouping attributes.

5.4 Spatial Operations

The temporal extensions with the bitemporal data streams described above and in the previous chapters do not require spatial operations. Nevertheless, spatial operations can be used with this approach and are important for moving object data streams. Odysseus already has spatial operations built in. These operations are lifted to the temporal domain just as all other existing non-temporal operations (see 5.2).

The spatial implementations in Odysseus that largely already existed before this work can be found in the bundle `de.uniol.inf.is.odysseus.spatial`. The spatial functions such as `SpatialWithin` and `FromWKT` as well as the spatial data types such as `SpatialPoint` are based on the JTS Topology Suite[4]. This standard Java library handles most spatial operations in Odysseus.

The spatial operations are map operations, i. e., they are not used with a special operator, but can be used in expressions. Hence, they are handled just like every other expression when using it with a prediction time dimension (see 5.2.6).

For some additional spatial calculations, the GeoTools library is used[5] (which internally also uses the JTS). For example, it offers a Coordinate Reference System (CRS) library, so that different European Petroleum Survey Group Geodesy (EPSG) numbers can be used in Odysseus. The standard CRS used is “urn:ogc:def:crs:epsg:7.1:4326” (“EPSG:4326”), i. e., WGS 84.

Listing 5.16 gives an example map operator that uses the `SpatialWithin` function. For each incoming stream element it calculates if the spatial point in the attribute “someSpatialPoint” is within the area in the attribute “area”.

[4] `https://github.com/locationtech/jts`
[5] `http://www.geotools.org/`

```
MAP({
    expressions = [
      ['SpatialWithin(someSpatialPoint, area)','isInside']
                  ],
    keepinput = true
},input)
```

Listing 5.16: An expression with a spatial function in a map operator

All spatial objects are represented by the `GeometryWrapper` class, which holds a JTS geometry and adds some additional, Odysseus specific interfaces. Hence, there is no extra class for spatial points, regions and so on, but only one class that represents all geometries.

5.5 Spatio-Temporal Filtering

The filter and refine concept is described in Section 4.8. An advantage of the concept is that it can be mainly implemented using existing DSMS functions. In this section, the implementation of the different filtering approaches is described.

5.5.1 Approximate Distance

The steps of this filtering approach are mainly the calculation of the possibly traveled distance of both involved moving objects, the calculation of the current distance between these objects and then the filtering using these results.

Maximum Traveled Distance

The approximation of the maximum traveled distance can be purely expressed with an expression, using the prediction time to which the moving object will be predicted, the record time at which the known location has been recorded and the maximum speed of the moving objects. Additionally, the time units of the stream and the prediction times need to be known and converted to seconds (if the maximum speed is given in meters per second). Even though this can be calculated with an expression without new functions, the term is quite cluttered and difficult to read. The following expression could, for example, be used for this purpose:

```
1 ((toLong(elementAt(asList(elementAt(PredictionTimes,0)),1)) * ${
      SECONDS_TO_MS})
2  -recordTime)
3  * (${MAX_SPEED} / ${MS_TO_SECONDS})
```

Listing 5.17: An expression to calculate the maximum traveled distance of a moving object

As can be seen in Listing 5.17, the expression is cluttered due to many Odysseus-internal functions which are necessary to interpret the values in the correct way. Basically, in Line 1, the end timestamp of the first prediction time interval is read (assuming that the prediction time only contains one time interval). It is multiplied by 1000 (variable `SECONDS_TO_MS`) to convert it from seconds to milliseconds (because the stream time is in milliseconds and the prediction time in seconds, in this case). This value is subtracted by the time when the last location was recorded (Line 2). To simplify this expression, it is assumed that the prediction time is always bigger than the record time. Now we have the temporal distance between the last known location and the predicted location. This is multiplied by the maximum possible speed of the object (Line 3) divided by 1000 (variable `MS_TO_SECONDS`) to convert the result into meters.

To simplify this process for the user, a simpler function is implemented, which automatically handles cases with, for example, multiple prediction time intervals, different time units and cases where the prediction is before the record time. The function `CalculateMaxTraveledDistance` expects three attributes: (1) the prediction times metadata, (2) the record time and (3) the maximum possible speed of the moving object. Then, the function can be applied as in Listing 5.18:

```
1 CalculateMaxTraveledDistance(PredictionTimes, recordTime, ${MAX_SPEED
      })
```

Listing 5.18: Using a function in an expression to calculate the maximum traveled distance of a moving object

Distance Between Moving Objects

The distance calculation is the next step of this approach. Distance calculations on the surface of the earth are more computationally expensive than a distance calculation in the two-dimensional Euclidean space. The costs of such a calculation depends on the needed accuracy. While the Vincenty's formulae [Vin75, Kar13] for distance calculations on an ellipsoid and the haversine formula or the spherical law of cosines [HRH+09] for spherical distance calculations need a number of trigonometric functions, the Euclidean dis-

tance calculation is less accurate on the surface on the earth, but needs less trigonometric functions.

Avoiding the more expensive calculations can improve the performance of the filtering approach. Therefore, in this step, an approximate distance calculation can be sufficient; the exact distance can later be calculated in the refinement step for the candidate objects. If an exact calculation for the distance is already desired in this step, it can also be applied using the standard spatial distance calculation function `OrthodromicDistance`.

For the approximate distance, the new function `ApproximateDistance` can be applied. It uses the Pythagoras formula to calculate the distance on a equirectangular projection, i. e., all rectangles on the earth map have the same size[6]. It is less accurate than a spherical or orthodromic distance calculation, but less computationally expensive [HHD12]. The accuracy differs depending on the location, distance and direction of the points. Figures 5.4 visualizes the relative error for the approximate distance in comparison to the haversine distance calculation for a scenario with distances up to about 5 km at the 53rd parallel north (one point is exactly at (53,0), the other at the respective coordinates of the x and y axis). The graph has been re-created and evaluated roughly based on the code from [Sal14]. Figure 7.3 in the appendix shows the absolute error for the same configuration.

The error, relative or absolute, is depicted by the colors. The circle shows the distance of 5 km between the two points. The error is for this scenario stays under 0.1 cm, respectively 0.00001 %. The error increases with bigger distances, as can be seen in Figures 7.4 and 7.5, but stays below one meter for distances up to 50 km (orange circle). Nevertheless, the error is bigger near the poles, as the Figures 7.1 (height of Spitsbergen) and 7.2 show.

The `ApproximateDistance` function can be used in an expression as shown in Listing 5.19:

```
1 ApproximateDistance(recordPoint, center_recordPoint)
```

Listing 5.19: Expression to calculate the approximate distance between two objects.

Tuple Filtering

Based on these calculations, a select operator can be used to filter out those elements which are too far from each other. The select operator is shown in Listing 5.20:

[6] For a closer look on this approach and other distance calculations, `https://www.movable-type.co.uk/scripts/latlong.html` offers detailed explanations and formulas.

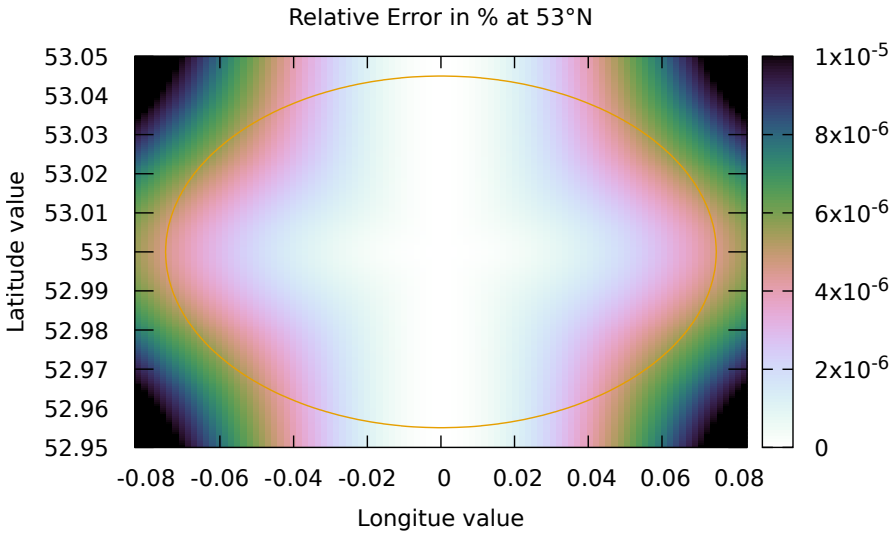


Figure 5.4: Relative error of approximate distance calculation compared to distance calculation with the haversine formula. Circle marks 5 km radius. Figure is reproduced and slightly modified from [Sal14].

```
SELECT({predicate = 'approxDistance < (maxTravelDistance + radius)'},
  input)
```

Listing 5.20: Select operator to filter out elements that are too far away.

`maxTravelDistance` is the estimated maximum possible distance that the two objects can travel and `approxDistance` is the approximate distance between the two objects at their respective last known locations. The `radius` is the distance of the radius query.

5.5.2 Single Rectangle

For the single-rectangle method (cf. 4.8.2.2), a new `createRectangle` function helps to create a rectangle around a point. It takes a center point and a distance around that point that needs to be within the rectangle. The rectangle can then be created with an expression in a map operator, as shown in Listing 5.21:

```
MAP({
  expressions = [
    ['createRectangle(center_recordPoint,${REAL_DISTANCE} + ${
        EXTRA_FILTER_DISTANCE})','RectangleFilter']
  ],
  keepinput = true
},input)
```

Listing 5.21: Creating a rectangle around a point.

In this example, the radius of the radius query is in the constant `REAL_DISTANCE`. The additional margin around the real distance is in the constant `EXTRA_FILTER_DISTANCE`. It is important to apply this map function before the join to reduce the number of calculations. The rectangle needs only to be calculated once for each new center element, not once for every new join output. Later, a select operator can be used to remove the elements which are not in the rectangle, as shown in Listing 5.22:

```
SELECT({predicate = 'SpatialContains(RectangleFilter,recordPoint)'},
  input)
```

Listing 5.22: Select operator to filter out elements that are not within the rectangle.

5.5.3 Multi Rectangle

The multi rectangle approach combines the two previous approaches and can therefore also reuse the functions from these methods. Instead of one, multiple rectangles are created, as shown in Listing 5.23:

```
addFilterAreas = MAP({
  expressions = [
    ['createRectangle(center_recordPoint,${REAL_DISTANCE} + 1 * ${
        FILTER_STEPS_METERS})','RectangleFilter1'],
    ['createRectangle(center_recordPoint,${REAL_DISTANCE} + 2 * ${
        FILTER_STEPS_METERS})','RectangleFilter2'],
    ['createRectangle(center_recordPoint,${REAL_DISTANCE} + 3 * ${
        FILTER_STEPS_METERS})','RectangleFilter3']
  ],
  keepinput = true
}, input)

filtersToList = MAP({
  expressions = [['toList(RectangleFilter1,RectangleFilter2,
      RectangleFilter3)','FilterAreas']],
  keepinput = true
}, addFilterAreas)
```

Listing 5.23: Creating multiple rectangles around a point.

Here, the margin around the real distance is increased by one additional `FILTER_STEPS_METERS` at each step. The rectangles are then put into a list to access them in the filter step later in the process. The actual filter step again uses the `CalculateMaxTraveledDistance` function (cf. 5.5.1). The whole filter step using the list of rectangles is shown in Listing 5.24:

```
1  /// Make filter visible
2  filterPreparation1 = MAP({
3    expressions = [
4      ['CalculateMaxTraveledDistance(PredictionTimes, recordTime, ${
           MAX_SPEED})','maxDistanceOther'],
5      ['CalculateMaxTraveledDistance(PredictionTimes, center_recordTime
           , ${MAX_SPEED})','maxDistanceCenter']
6    ],
7    keepinput = true
8  }, predTime)
9
10 /// Calculate the bucket
11 filterPreparation2 = MAP({
12   expressions = [['min(toLong(Ceil((maxDistanceOther +
         maxDistanceCenter) / ${FILTER_STEPS_METERS})-1), ${NUMBER_STEPS
         }-1)','AreaBucket']],
13   keepinput = true
14 }, filterPreparation1)
15
16 /// Filter step
17 filterStep = SELECT({
18   predicate = 'SpatialContains(elementAt(FilterAreas, AreaBucket),
         recordPoint)'
19 }, filterPreparation2)
```

Listing 5.24: Filter elements with multiple rectangles based on the temporal distance.

The map operator in Line 2 calculates the maximum traveled distance of both involved moving objects. The following operator in Line 11 calculates, which of the rectangles need to be used for this stream element, i. e., pair of center element and other element. When they can only have traveled a short distance, a smaller rectangle will be used. If they can have traveled a long distance, a bigger rectangle will be used. If they can have traveled further than the biggest rectangle, the biggest rectangle will be used (therefore the `min`).

In the last operator in Line 17, the filter is applied by choosing the correct rectangle from the list (`elementAt(FilterAreas, AreaBucket)`) and calculating if the element is within the respective rectangle (`SpatialContains`).

The next step can be the the calculation of the real distance and the selection based on the result. These steps would then be done in the temporal domain, i. e., calculating the distances for the time instances in the `PredictionTimes` time intervals.

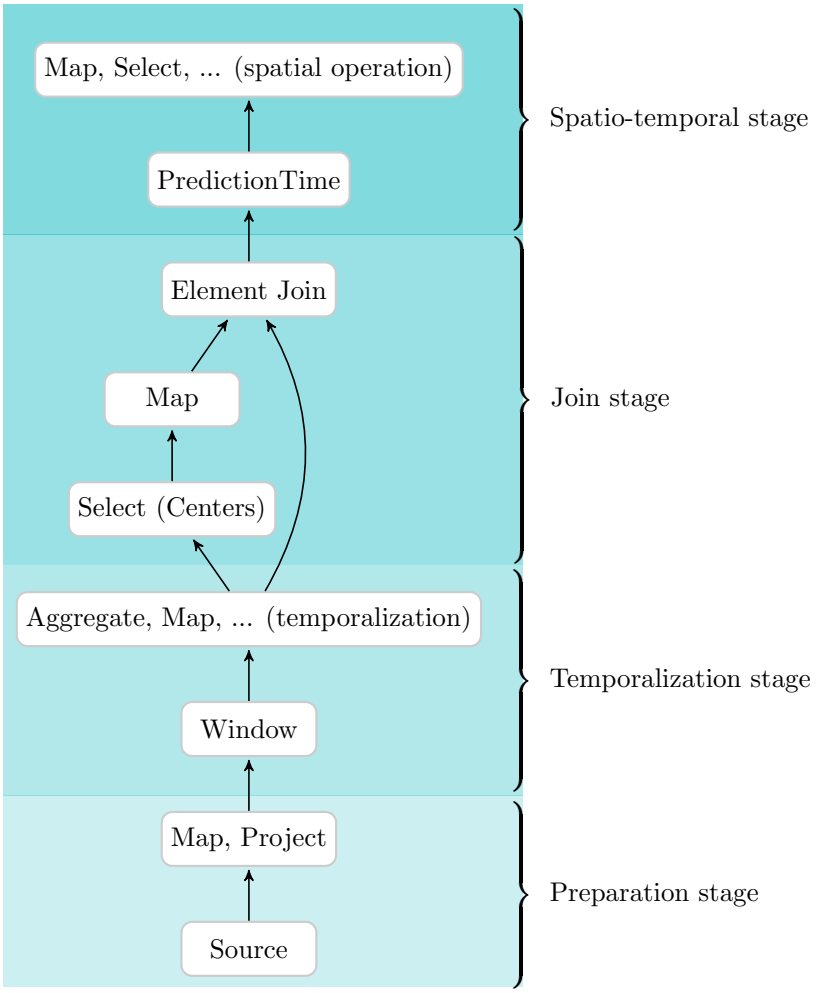


Figure 5.5: Typical structure of a spatial query with temporal functions [BG19]

5.6 Generic Moving Object Query Structure

The previously presented implementations can be applied to queries with moving objects. Those queries can be structured depending on the use case and can be build very differently. Nevertheless, for some common use cases that need to combine multiple moving objects, a generic structure can help to build the queries. Such a generic structure can ease the development and gives an idea on how to start more specific queries.

Figure 5.5 shows a query plan for temporal queries on moving objects. The plan is divided into typical stages of a spatio-temporal query working on a stream with multiple moving objects.

The first stage on the bottom is the preparation stage. The incoming stream is converted into a common format. For example, latitude and longitude values are packed into an internal `point` representation. The preparation can typically be done with map and project operators. The next step is the creation of a temporal attribute for the representation of the object's movement. Here, the incoming location updates from the moving objects are used to create a temporal spatial object, e. g., a temporal point. When using an aggregation operator for this purpose, a window operator should be used before it to limit the used location updates to the newest ones. Other means of creating a temporal attribute are also possible at this stage (cf. Section 4.2.2).

The join stage after the temporalization step defines how the moving objects are combined for the spatial operations later in the query. A route or select operator can be used to split the stream, which initially contains the movements from all objects. When having more than one center element, the second input with the non-center objects in fact has to contain the center objects, too. If this would not be the case, the join would not join two center objects, which could lead to missing results. To avoid a center object being joined with itself, the predicate of the join has to ensure that the `id`s of the two joined objects are different.

Before joining the two streams again to have two temporal attributes (typically two temporal points) in one stream element, it is useful to rename at least one of them to distinguish them later on in the query. That is why another map operator is added in-between the route and the join. The join operator can use the built-in element window to join only the latest elements.

The last stage is the spatio-temporal stage. Here, the user can define the prediction time with the `PredictionTime` operator. The last step is the spatial operation the user wants to apply. This can be everything offered by standard operators, for example map operations on the temporal attributes, e. g., the temporal points.

This query structure can be used for different purposes. The example scenarios in Chapter 6 use the same basic structure.

5.7 Summary

The implementation described in this chapter finalizes the progress from the logical concept to the implementation, following the physical concept in-between. First, the architecture of the underlying system for the implementation is explained. Odysseus, as the DSMS that is extended by this thesis, is presented in Section 5.1. Section 5.2 continues with the description of the implementation of the temporal types into Odysseus, which is a main part of this thesis. It contains, among others, the integration of the new `PredictionTimes` metadata and the temporal attributes.

The temporalization functions as well as the temporal implementations for the standard operators, such as map (cf. 5.2.6) and select (cf. 5.2.8), are also explained in this

section. These implementations have in common that they build on the existing non-temporal implementations and only extend them. That way, it is not necessary to implement the existing logic again.

The extension of the metadata by the `PredictionTimes` as well as the temporal trust value (cf. 4.5) also needs to be implemented, even though it also builds upon existing solutions for metadata handling. For example, the merging process for prediction time intervals with different granularities is described in Section 5.2.13 and the merging process of temporal trust values in Section 5.2.14.

The element join is an extension which is not connected to the temporal types, but is necessary for the queries on moving objects to avoid unnecessary latencies. The implementation of the extension of the existing join operator is described in Section 5.3. A short overview of the already existing spatial operations in Odysseus is given in Section 5.4.

Section 5.5 continues the chapter with the description of the implementation of the spatio-temporal filtering approach. The implementation of the three different approaches with existing and new functions of Odysseus are explained, so that these approaches can be applied in a query.

Finally, Section 5.6 describes a generic structure for moving object queries. It can be used as a pattern when creating new queries on moving object data streams with temporal attributes and eases the development of such queries.

This chapter finalizes the description of the development of this thesis. The implemented system is evaluated in the next chapter to show the capabilities of the approach and the performance of the implementation, as well as the effect of the filtering approaches on the result quality and performance.

6 Evaluation

The initial goal of this thesis is the research question from Chapter 1. The question asks for two things: (1) expressing, i. e. defining, queries on moving object data streams and (2) processing these queries. The concept uses approaches from data stream processing and moving object databases, i. e. the moving object algebra, to tackle these questions. Chapter 5 covers the feasibility evaluation by explaining how the concept can be implemented into a DSMS. This chapter describes the next step of the evaluation and uses the implementation from the previous chapter to evaluate the new concepts.

Section 6.1 describes the data that is used for the evaluation. The data that is used in most evaluations is from the maritime domain. Using this data has some advantages for the evaluation. First, it has typical properties of data from moving objects. The vessels are not (strictly) bound to a network, as it would, for example, be the case for trains. Second, they send their location every few seconds to minutes via AIS, wherefore the data has typical data stream characteristics and is broadly available. Using this data does not limit the generality of the evaluation, as the general characteristics are domain agnostic. The generic nature of the approach is additionally demonstrated with an additional scenario outside of the maritime and outside of the moving object domain in Section 6.2.8 with an energy consumption scenario.

This data is used for the scenario and performance evaluations. The scenario evaluation in Section 6.2 describes different scenarios, mainly from the maritime domain, and explains how they can be realized with the implemented system. This part of the evaluation evaluates the first part of the research question: “How can queries on spatio-temporal data streams from moving objects be **expressed flexibly and with generic semantics** [...]” (cf. 1.2). It does so by explaining the implemented system with common moving object queries and by generalizing from these specific queries. Due to the generalization and the overall more generic approach to the queries, the evaluation, even though using maritime scenarios, does not limit the validity to this specific domain.

Section 6.3 uses the described data and queries to evaluate the performance of the implemented concept and with that the second part of the research question: “[...] and **processed efficiently?**” (cf. 1.2). It analyses how queries can be build with different performance goals and how the analyzed data affects the performance of the queries. Additionally, the filter and refine approaches are evaluated to measure the effect on the query performance and to prove that a streaming approach with a DSMS can be the foundation for efficient query processing.

6.1 Data Description

For both the scenarios and the performance evaluation, data from the maritime domain in the form of AIS data is used. As mentioned above, it has typical properties of data from

moving objects. The locations of the vessels are provided at irregular time intervals as latitude and longitude values together with an identification number and other attributes. Additionally, it is not guaranteed that a vessel will send its location forever; it could also be the case that the vessel leaves the observed area and will not send data again.

For the performance evaluation, AIS data is used. The platform "marinecadastre.org" offers AIS data from the US coast over several years and with that a vast amount of data for evaluation[1]. The data contains comma-separated values (CSV) files with the stored data, which then can be utilized as if it would be a stream. The data contains the timestamp when the message was received, which can be set as the stream timestamp.

Unfortunately, the base stations, which receive the AIS messages and set the timestamp, seem to not have synchronized clocks, wherefore the messages are not correctly ordered. In a real streaming scenario this is unlikely to happen: when messages are received by the base station, they should be immediately forwarded to the central DSMS. Then, the timestamp can be set by the DSMS to the point in time when the message was received. The handling of unordered streams is a complex issue in data stream management without a perfect solution but with multiple proposed approaches to handle such streams [SW04, LTS+08, ABC+15, CKE+15b]. One possible approach could be to wait a minute or so and sort the incoming elements within this time window. Unordered elements that arrive afterwards with a higher time difference (i. e., more than one minute in this example) would be dropped. Nevertheless, this approach makes the query lag behind the current stream by the time it waits for temporal outliers, i. e., one minute in this case. To avoid this problem for this data, the provided data set is sorted beforehand.

The locations are stored in latitude and longitude values with a WGS 84 datum[2]. For the evaluation, the data set from April 2017 from UTM zone 10 is used, but trimmed to the data points from April the 1st 2017, 10:00 to 12:00 o'clock to allow better data handling when analyzing the query results. The data contains 81 132 AIS messages from 1 171 vessels, which are in average around 69 messages per vessel and roughly one message per vessel every two minutes. The original data set is already filtered to at maximum one message per minute per vessel. Figure 6.1 shows the data drawn on a map. The borders of the UTM zone 10 are visible where the data points end in the west and east.

For one scenario that is not in the domain of moving objects, an artificial data set from the energy domain is used, which is described in Section 6.2.8.

6.2 Scenario Evaluation

In this section, selected queries on moving object data streams are presented to show how to apply the concept to real world data and scenarios. The radius- or distance

[1] https://marinecadastre.gov/ais/
[2] https://inport.nmfs.noaa.gov/inport/item/53161

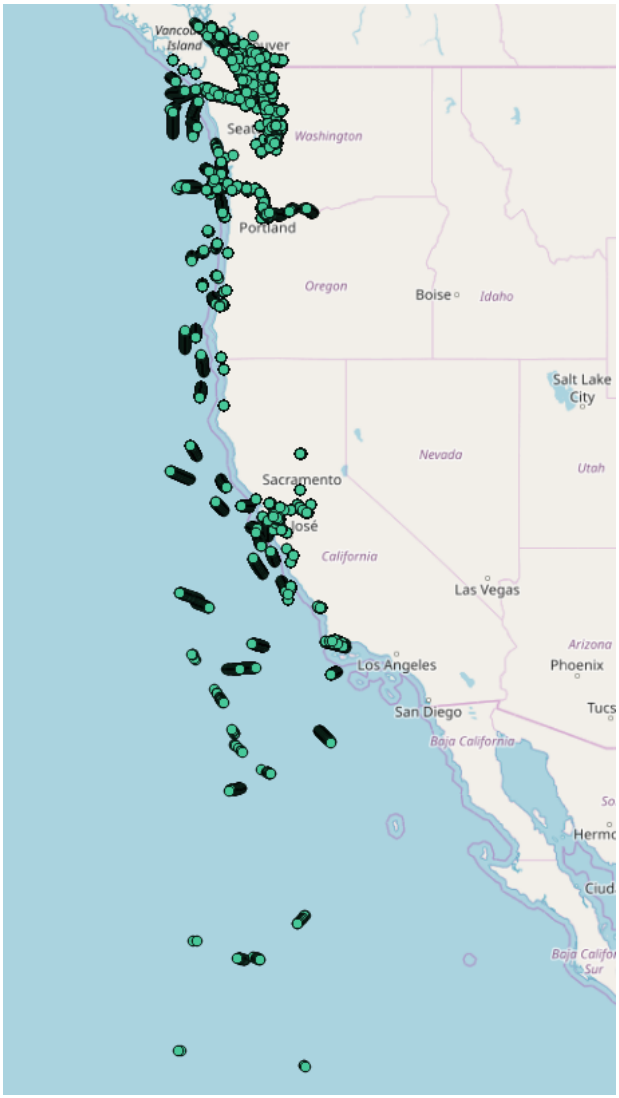


Figure 6.1: All data points from the data set in UTM zone 10 for April the 1st 2017 between 10:00 and 12:00 o'clock on a map. Map background: OpenStreetMap

query [Bri07] as well as the kNN query are typical queries in the spatial domain [RSV02, ZZP+03, HXL05, Bri07, ZJDR10]. Even though their results seems to be similar (the objects in a certain radius around a point or a certain number of nearest objects around a point), the approach to these two queries is different. Therefore, it is interesting to see how these queries can be implemented with the presented solution for streaming queries and how they differ from each other. These two queries are followed by a moving region query (cf. Section 6.2.3). In contrast to the radius and kNN queries, which in the given examples work on points, the region query adds the aspect of regions and shows the applicability of the concept to other geospatial objects than points. These queries build upon the generic query structure presented in Section 5.6.

Following the usage of maritime data, two common maritime queries are also part of the evaluation. The Closest Point of Approach (CPA) and Closest Time of Approach (CTA) queries in Section 6.2.4 expand the evaluation to more specific queries, in addition to the more general queries motivated above. The following scenarios go into more detailed aspects of the concept and implemented solution. In Section 6.2.5, a query with a specific function from the moving object algebra is presented. It shows the `trajectory` function as a representative function that converts temporal attributes to non-temporal attributes.

While the radius and kNN queries of this evaluation use an aggregation function to temporalize the location attribute, the scenario in Section 6.2.6 shows how a predefined route can be converted into a temporal attribute. Section 6.2.7 looks at the newly introduced metadata for the temporal trust and shows how the trust estimation can be applied.

Finally, Section 6.2.8 steps back from moving object data and shows that the presented approach of this thesis is also applicable for other scenarios. In the given example, the energy consumption of households is converted to a temporal attribute to have more up-to-date values.

6.2.1 Radius Query

A radius query (also called distance query [Bri07]) is a common spatial query. The goal is to find all objects within a certain distance around an object. In the case of moving object data streams, the objects of interest are moving, i. e., they continuously change their location [BG19]. For each calculation of new results, the most recent information from each moving object shall be used. There are different possible solutions for this query, with different window settings and joining predicates. One possible solution is depicted in Figure 6.2.

Similar to the query depicted in Figure 4.13 in Section 4.6.2, this query receives the data from an external source with a certain schema. In this case, the data has the schema described in Section 6.1. The initial map operator at the bottom prepares the schema for the rest of the query, for example, converting latitude and longitude values into an

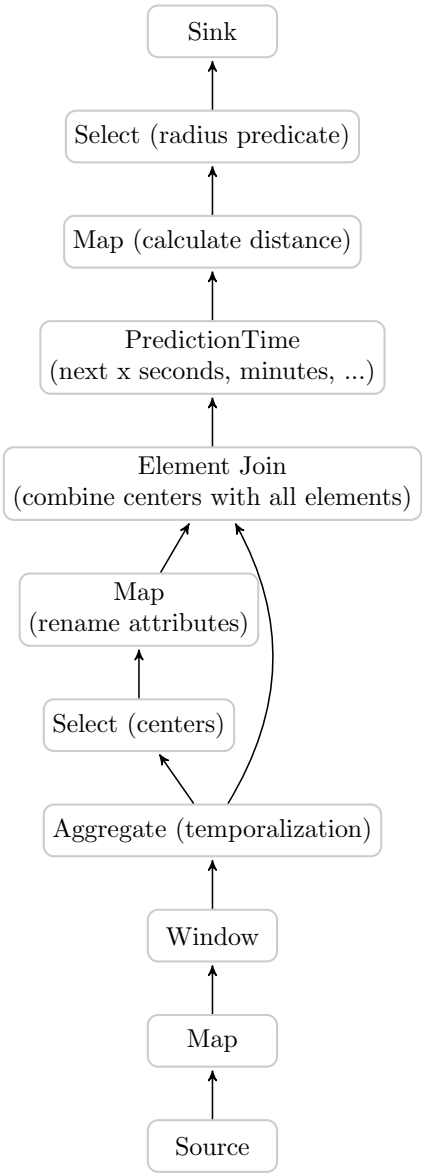


Figure 6.2: Query plan for a radius query

internal point representation. The following window is necessary for the aggregation operation, which in this case is used to convert the non-temporal points to temporal points (cf. Section 4.2.1). The aggregation can, for example, use a function with linear interpolation between the known points. The definition of the aggregation operator can be seen in Listing 6.1. The operator uses the function `ToLinearTemporalPoint`, which creates a temporal point with linear interpolation using the non-temporal `SpatialPoint` attribute. It writes the results to the attribute `tempSpatialPoint`. The function is applied to each vessel individually, which is important to not mix locations from different vessels into one temporal point. This is achieved by grouping the incoming stream elements by their `id`.

Another option used here is `eval_at_outdating`, which is set to `false`. This means that no new output is generated when a stream element leaves the window. If that option would not be set to `false`, potentially multiple outputs are generated for the same moving object. Semantically, this would also be correct. Nevertheless, it would fill the stream with additional results, which do not provide a real benefit for this use case, as they are not based on new location updates. Finally, in Line 9 the previous map operator is defined as the input for the aggregation operator.

```
1 temporalizationAggregation = AGGREGATION({
2   aggregations = [[
3     'function' = 'ToLinearTemporalPoint',
4     'input_attributes' = 'SpatialPoint',
5     'output_attributes' = 'tempSpatialPoint'
6   ]],
7   group_by = ['id'],
8   eval_at_outdating = false},
9   map)
```

Listing 6.1: Aggregation operator to convert a point to a temporal point with linear interpolation

The select operator separates the center objects (i. e., the objects for which the objects in their surroundings are searched for) from the other objects, so that they can be joined afterwards. The number of center objects depends on the predicate. Typically, they are chosen by their `id`. It is also possible to join all objects with each other to do the radius query for all objects. A sample for a select operator is shown in Listing 6.5. The predicate chooses four moving objects with the ids being “1” to “4”. These are the observed center objects for which the other objects within a certain distance will be calculated. Another notable option is the `heartbeatrate = 1`. This is used for the optimized element join (cf. Section 4.7.2) and tells the join operator the temporal advance on this side of the stream even if all elements are filtered out.

```
/// Select the center
selectCenter = SELECT({
  predicate = 'contains(id, toList(1,2,3,4))',
  heartbeatrate = 1
},input)
```

Listing 6.2: Select operator to chose the center objects

The map in-between the select and the join only renames the attributes `id` and `tempSpatialPoint` to `id_center` and `tempSpatialPoint_center` to distinguish the center attributes from the non-center attributes in the join. The following join operator is an element join, which has its internal element window (cf. Section 4.7.1). It can be configured as in Listing 6.3. Again, it is important to group the element window, as we want the newest location information from each vessel, not only the newest location information that has been received in general.

```
JOIN({
  elementsizeport0 = 1,
  elementsizeport1 = 1,
  group_by_port_0 = ['id_center'],
  group_by_port_1 = ['id']
  }, renameCenter, temporalizationAggregation)
```

Listing 6.3: Join operator to join center objects with other objects

Next, the prediction time is defined. This can be done with the `PredictionTime` operator. The prediction time determines for which points in time the expressions with temporal attributes, in this case the temporal point, will be calculated. Having the base time unit set to seconds, let us choose two seconds here. At this point, the actual prediction time is less important. Here, two seconds simply have the effect that the result is calculated for two different points in time. Later in this chapter, the effect on the performance of different prediction times is measured. The definition of the operator can be seen in Listing 6.4. In this case, the prediction time is defined in relation to the start timestamp of the stream time.

```
PREDICTIONTIME({
  addtostartvalue = [0, 'seconds'],
  addtoendvalue = [2,'seconds']},
  join)
```

Listing 6.4: Operator to set the prediction time of the joined stream elements

Now, the spatial query can be run on the resulting stream. Spatial expressions will automatically be evaluated in the temporal dimension defined in the `PredictionTime` operator. Hence, the definition of the expressions is identical to a query that does not use any temporal attributes. In Listing 6.5, two operators are defined: (1) the `map` operator to calculate the distance and (2) the following `select` operator to filter out elements which are too far away. The distance is calculated using the `OrthodromicDistance` function, which is based on the JTS method of the same name. The result is a temporal double, which is used in the `select` operator. The `keepinput = true` option is used to avoid the other attributes in the stream element besides the result of the expression to disappear. The predicate in the `select` operator filters out all elements which are further away than 5000 meters. The `sink` operator can write these results in a file, a database or send it to another system.

```
1 calcDistance = MAP({
2   expressions = [['OrthodromicDistance(center_SpatialPoint,
        SpatialPoint)','tdistance']],
3   keepinput = true
4   },input)
5
6 SELECT({
7   predicate = 'tdistance < 5000'
8   },calcDistance)
```

Listing 6.5: Select operator based on a distance calculation on temporal attributes

6.2.2 kNN Query

A kNN query searches for the k objects that are closest to a center object. It is, next to the previously described radius query, a common query in the spatial domain [EM13, YWS15, KS04a]. Even though it seems to be very similar, the queries differ processing-wise. While the radius query can be implemented using a select operator, the kNN query needs to be solved using an aggregation. Therefore, the spatio-temporal stage of the generic query structure (cf. Section 5.6) differs from the radius query. Nevertheless, it also shows that the mentioned query structure is generic enough to be used for these different queries.

In this scenario, all objects are moving. The spatial kNN operation can be implemented by a distance calculation in a `map` operator, which is identical to the radius query (cf. Section 6.2.1) and a subsequent TopK operation in an `aggregation` operator, which differs from the radius query. The TopK operation calculates the k values that have the highest (or lowest) score according to a scoring attribute. In this case, the temporal distance is used as the scoring attribute.

```
AGGREGATION({
  aggregations = [[
    'FUNCTION' = 'TopK',
    'TOP_K' = '2',
    'INPUT_ATTRIBUTES' = ['id','tdistance'],
    'SCORING_ATTRIBUTES' = 'tdistance',
    'UNIQUE_ATTR'='id',
    'descending' = false,
    'always_output' = true
  ]],
  group_by = ['id_center']
},calculateDistance)
```

Listing 6.6: TopK function in aggregation operator to calculate kNN

The PQL definition of the aggregation operator is shown in Listing 6.6. The `INPUT_ATTRIBUTES` option is used to define which attributes are in the output element. If, for example, only the ids of the closest elements are needed, the "tdistance" could be removed here. The `UNIQUE_ATTR` option makes the aggregation function replace a previous stream element with the same id by a new one. This mimics an internal element window of size one and is needed here to not have the same element as one of the nearest neighbors multiple times.

Another difference to the radius query is the prediction time definition at the end of the query. An additional time window is added after the join, because the element join removed the end timestamp (cf. Section 4.7.1). Without a time window, the aggregation operator would keep all elements in its memory, which could lead to a memory overflow in long-running queries.

Additionally, the prediction time intervals are aligned so that all prediction time intervals within a window are equal. Having this, the merging function for the prediction time intervals in the aggregation (cf. Section 4.3.5) can be freely chosen, as both the union and the intersection merge function would have the same result. The alignment is achieved with a tumbling time window for the stream time and an option in the `PredictionTime` operator to align the prediction times at the end timestamp of the stream time interval.

Listing 6.7 gives an example on how this can be achieved using PQL. The important option in the `PredictionTime` operator is `alignAtEnd`. Using this option, the times from `addToStartValue` and `addToEndValue` are added to the end timestamp of the stream time. Due to the previous tumbling window, the end timestamps of one window are all equal, wherefore the prediction time intervals for all stream elements within the window are identical. The window alignment is depicted in Figure 6.3. The figure shows the stream time intervals of the stream elements. They have different start timestamps, but are, due to the tumbling window, aligned to one end timestamp per window.

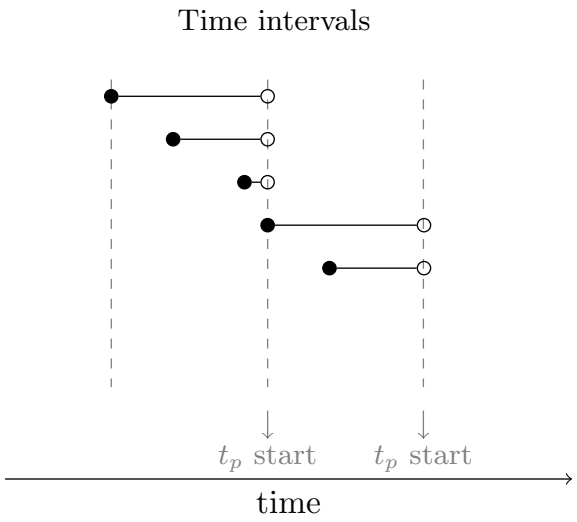


Figure 6.3: Tumbling window for prediction time alignment before an aggregation

The figure shows two windows, i. e., two different end timestamps for the five stream elements. The prediction time interval is aligned at the end timestamp, wherefore all stream elements in one tumbling window have the same prediction time interval.

Effectively, the query looks up to 61 seconds into the future, which happens exactly when the first element of a new tumbling window flows into the window. When the last element of the tumbling window flows in, the query looks only one second into the future. The advantage of this approach is that there is at least one point in time where all location updates of the last 60 seconds (or another time span, depending on the time window) are used for the TopK calculation. In a sliding window with either of the two merging functions, overlapping time intervals for the prediction time intervals either cannot be guaranteed (for the intersection merge) or at some points in the prediction time, the aggregation calculates a nearest neighbor without using all available moving objects (for the union merge). Due to these limitations, a tumbling window should be used in this case.

```
alignWindow = TIMEWINDOW({
    size = [60, 'SECONDS'],
    advance = [60, 'SECONDS']
  },recombine)

/// Set the prediction time
PREDICTIONTIME({
    addToStartValue = [0, 'seconds'],
    addToEndValue = [2, 'seconds'],
    predictionBaseTimeUnit = 'SECONDS',
    alignAtEnd = true
  },alignWindow)
```

Listing 6.7: A tumbling stream time window to align the prediction time intervals

This scenario shows how customizable the prediction approach with the time intervals is, but also illustrates the complexity that can arise when developing the correct query for a certain use case. There are multiple other possibilities to develop a kNN query. Especially the handling of the temporal dimensions (both stream and prediction time) can be adapted to fit to the needs of the user.

6.2.3 Moving Region Query

Moving objects are often represented by moving points, which are in many cases a close-enough representation and reduce complexity in queries and calculations. Nevertheless, some use cases require moving objects to have a spatial extend, for example, if a moving oil field, a storm or a plastic spill needs to be represented [BG19]. Representing regions within attributes of a stream object is possible due to the use of the Simple Feature Access model for geometries (cf. Section 2.4). With this scenario, the possibility to use other geometries than points is shown. New aspects are, among others, the prediction of a region instead of a point and other spatial operations than distance calculations, for example, the `spatialWithin` operation.

Figure 6.4 depicts an example for a moving region query. In this example, a moving point enters a moving region (a polygon), is within the region for a while before the point leaves the region again.

As with moving points, for a moving region a temporal function is necessary to represent the movement. Such a temporal function can, for example, be achieved with a temporalization function. Snapshots of a moving region are used to interpolate and extrapolate the movement of the region between the snapshots and for predicting future movement. The demonstration functions in this example only support moving regions that do not change the number of vertices over time. The vertices are internally represented as individually moving points.

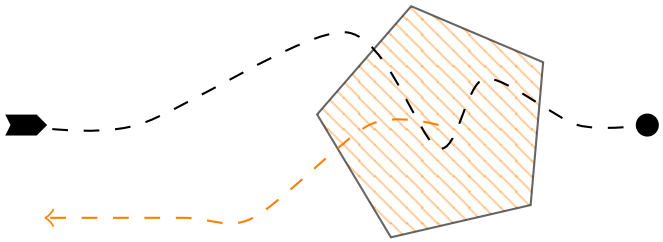


Figure 6.4: A moving region (dashed polygon) and a moving point

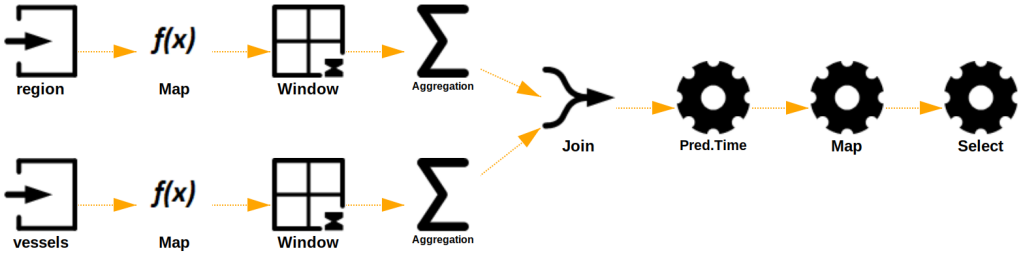


Figure 6.5: Query plan view in Odysseus showing a moving region query

An example query plan with a moving region and moving points is shown in Figure 6.5. In contrast to the other examples, this query has two input sources: one on the top for the moving region (i. e., the oil spill) and one on the bottom for the moving objects (i. e., the vessels). Both are converted into temporal spatial objects with a temporalization function in an aggregation operator and then joined. Again, an element join is used to combine the newest moving region with the newest moving point from each vessel. The result is a stream with stream elements containing a temporal point and a temporal region, among other attributes. Next, the prediction time is set, for example, to the next two seconds or minutes, depending on the use case.

The map and select operators at the end of the query do the spatial calculation: the map operator checks, if the spatial point is inside of the spatial region at the prediction time intervals given the previous PredictionTime operator. The select operator filters out all the vessels which are not in that region (or filters objects which are only partly in that region to the respective time intervals).

This query demonstrates that the concept also works with other spatial objects than points. The necessary extensions are a temporalization function and an internal representation for moving regions. Both can and should be adapted to the use case.

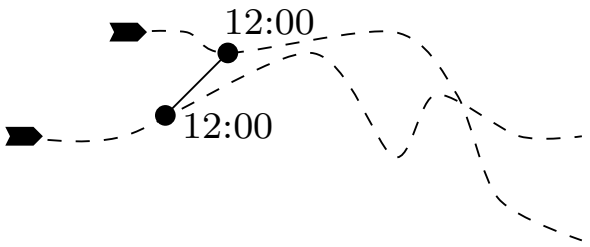


Figure 6.6: CPA and CTA of two moving points

6.2.4 Closest Point of Approach (CPA) and Closest Time of Approach (CTA) Query

A common query in the maritime domain is the Closest Point of Approach (CPA) or Closest Time of Approach (CTA) query [CD16, KWZH14]. Having two moving objects, the query calculates where or when these objects will be closest to each other. It is important to notice that this is not simply the minimum distance between two trajectories, but the minimum distance of two moving objects considering the time. An example is depicted in Figure 6.6. Two vessels move close to each other, from left to right, each on its own trajectory, which are depicted with dashed lines. The CPA is at 12:00. Further to the right, the trajectories are closer to each other, but the distance between the two objects, considering that the two objects are not at the narrow gap at the same time, is bigger.

The CPA query can be implemented with the `atMin` function from the moving object algebra. It returns the minimum value of a temporal type. A plan of the last operators of this query is depicted in Figure 6.7. The first operators are left out because they are equal to the generic query structure (cf. Section 5.6). Before the shown operators, two vessels for which the CPA should be calculated are picked and joined together so that their temporal points are in one stream element.

The first operator in the figure (`PredictionTime`) sets the prediction time before the distance calculation. If, for example, the CPA within the next 10 minutes is of interest, the prediction time can be set to include the time from "now" until in 10 minutes, maybe with a granularity of one second. Next, the distance for that time interval is calculated, resulting in a temporal double value.

The select operator uses the predicate `atMin(tdistance, PredictionTimes)`. The `PredictionTimes` point to the metadata field for the prediction time intervals, which in this case is necessary for Odysseus to access this field. The function reduces the `PredictionTimes` to the point(s) in time where the distance is at its minimum. This would be sufficient to have the CPA (the temporal points from the two vessels at the prediction time) and the CTA (the resulting prediction time). To make the result in the distance attribute (a temporal double), which is a `GenericTemporalType`, more readable, the

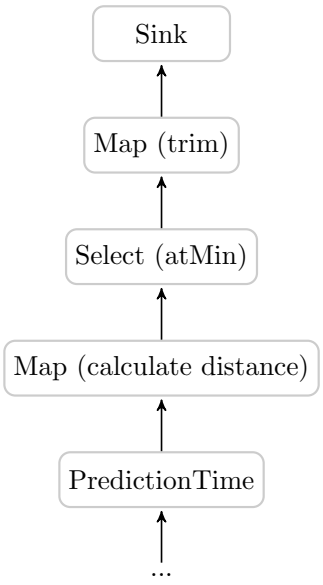


Figure 6.7: Last operators of a CPA query

other distance results are removed with the `trimTemporal` function (cf.5.2.3). The expression in the last map operator is `trimTemporal(tdistance,PredictionTimes)`.

Table 6.1 shows an example result for this query. The metadata fields have a slight gray background. The `prediction_time` field states that the minimum value is at point in time `1491005925` and that the granularity of the prediction is seconds. The distance between the vessels with the ids `366970430` and `367605150` (fields `id_1` and `id_2`) at this point in time is 9451.135691970458 meters (field `tdistance`). The temporal points are represented by a linear function, here simply represented by their speed and azimuth (fields `point_1` and `point_2`).

6.2.5 Trajectory Length of Moving Objects

The previous queries have in common that they start with non-temporal data streams, convert some attributes into temporal attributes and then continue to work with temporal attributes. Nevertheless, it can also be the case that temporal attributes are converted back into non-temporal attributes. The `trajectory` function from the moving object algebra [GBE+00] is such a function. It takes a temporal point as its input and creates a non-temporal spatial line. On this line, the length can be calculated to get the distance the moving object traveled during the prediction time.

Assuming that the attribute `tempSpatialPoint` is a temporal point, the operators in Listing 6.8 calculate the distance that will presumably be traveled in the next four minutes. The prediction time is set to the next four minutes, starting with the latest known

field	value
tdistance	GenericTemporalType: 1491005925000 = 9451.135691970458
id_1	366970430
id_2	367605150
point_1	tpoint: speed: 1.5069861260011568E-6 $\frac{m}{timeInstance}$; acceleration: 0.0 $\frac{m}{timeInstance^2}$; azimuth: 145.22315846095213°
point_2	tpoint: speed: 1.6975420708553307E-6m $\frac{m}{timeInstance}$; acceleration: 0.0 $\frac{m}{timeInstance^2}$; azimuth: -54.2592189608814°
stream_time	[1491005925000,∞)
prediction_time	SECONDS: [[1491005925, 1491005926)]

Table 6.1: Example result of a CPA query

point in time (Line 1). The first map operator in Line 7 creates a non-temporal `MultiLineString` (cf. Section 2.4) from the temporal point `tempSpatialPoint`. The `MultiLineString` is used here instead of a simple `LineString` because the prediction time can have multiple time intervals, which would result in multiple non-temporal trajectories, i. e., a `MultiLineString`.

The last map operator in Line 14 calculates the length of the non-temporal `MultiLineString`. The result is given in meters.

```
predTime = PREDICTIONTIME({
  addtostartvalue = [0, 'MINUTES'],
  addtoendvalue = [4, 'MINUTES'],
  predictionbasetimeunit = 'MINUTES'
},input)

calcTraj = MAP({
  expressions = [
    ['Trajectory(tempSpatialPoint, PredictionTimes)','traj']
  ],
  keepinput = true
},predTime)

MAP({
  expressions = [['SpatialLength(traj)','len']],
  keepinput = true
},calcTraj)
```

Listing 6.8: Calculating the traveled distance of a temporal point

6.2.6 Using Predefined Routes

Temporal attributes cannot only be created from non-temporal input data, such as the movement of an object, but also from attributes which already include temporal information about an object (cf. Section 5.2.5). For example, the movement of a point can be described in a GeoJSON-like text. In this scenario, we have the movement of one vessel from a predefined trajectory, e. g., created by a navigation system ("vessel 1"). The movement of other vessels is created by a temporalization function based on the past movement. The data is combined to calculate the distance between vessel 1 and the other vessels.

The source for the temporal GeoJSON is a string attribute containing the GeoJSON string. It can be converted to a temporal point using the `FromTemporalGeoJson` function as can be seen in Listing 6.9.

```
MAP({
  expressions = [['FromTemporalGeoJson(data)','tempTrajectory']]
},input)
```

Listing 6.9: Converting a GeoJSON string in the "data" attribute into a temporal point in the "tempTrajectory" attribute

field	value
tdistance	GenericTemporalType: 1491004910000 = 4733.573835538204, 1491004911000 = 4733.573835538204
id	3
stream_time	[1491004910000,oo)
prediction_time	SECONDS: [[1491004910, 1491004912)]
temporal_trust	GenericTemporalType: 1491004910000 = 1.0 1491004911000 = 0.14722182877031387

Table 6.2: Example result of a query using the temporal trust

After a join with the other temporal points from a different source, the temporal point can be used like every other temporal point, for example, to calculate the temporal distance between the moving objects.

6.2.7 Trust Estimation

The trust value (cf. Section 5.2.14) to estimate the trustworthiness of a predicted result is a temporal meta attribute. It can be activated by adding it to the `metaattribute` field of the access operator (cf. Listing 6.10). In the background, the function to convert attributes into temporal attributes will additionally create a function which estimates the trust. In fact, the trust function is always created when creating a temporal attribute but only added to the stream elements metadata if the trust metadata is activated.

```
1 metaattribute = ['TimeInterval', 'PredictionTimes', 'TemporalTrust']
```

Listing 6.10: Using the "TemporalTrust" meta attribute in an access operator

The calculated temporal trust value is added to the metadata attributes of the stream elements. Table 6.2 shows an example stream element with a temporal trust attribute. As can be seen, the trust value is very high at `1491004910000` and decreases to a lower value at `1491004911000`. That is because the trust values of different temporal attributes are merged. In this case, a map operator calculates the distance between two temporal points. Both have a high trust at the first point in time (in the prediction time dimension), but at least one has a lower trust at the second point in time. The merge function uses the lowest trust of all involved temporal attributes (cf 4.5).

6.2.8 Energy Consumption

While this thesis is mainly focused on spatio-temporal data, the general approach of temporal attributes can also be applied to other types of data. To give an example with data from a different domain, in this section a scenario from the electricity domain is described.

The scenario is the following: a smart meter measures the energy consumption of a household and sends an aggregated value with the total sum of the energy consumption every 15 minutes. Listing 6.11 shows such a data stream.

```
1  id,wh,time
2  1,0,0
3  2,0,3
4  3,0,6
5  1,115,15
6  2,50,18
7  3,250,21
8  1,200,30
9  2,60,33
10 3,500,36
11 1,210,45
12 2,100,48
13 3,600,51
```

Listing 6.11: Data stream with the `id` of the smart meter, the watt hours (`wh`) consumed so far, and the `time` in minutes when the data was measured

To have a more up-to-date view on the current energy consumption of the households, the consumption per minute per household shall be estimated for the next 15 minutes until the new measured value arrives. Higher values, e. g., more than 300 watts, shall be presented to the user, i. e., be the result of the query.

To achieve this goal, the energy consumption is converted into a temporal attribute. For example, a linear prediction function based on the last two measured values can be applied. This first part of the query is shown in Listing 6.12. The time window with a size of 20 minutes has the effect that there are exactly two valid measurements per smart meter in the current window. When a new element arrives 15 minutes after the last one, the previous one is still valid so that it will be used in the aggregation operator. The aggregation uses the last measurements and creates a temporal function and puts it in the attribute `temp_wh`.

```
1 /// Only use the last two measured values
2 time = TIMEWINDOW({size = [20, 'minutes']
3 },input)
4
5 /// Convert the wh-attribute to a temporal double
6 temporalize = AGGREGATION({
7   aggregations = [['function' = 'ToTemporalDouble', 'input_attributes
        ' = 'wh', 'output_attributes' = 'temp_wh']],
8   group_by = ['id'],
9   eval_at_outdating = false
10 },time)
```

Listing 6.12: A time window and the temporalization step in an aggregation operator

The middle part of the query, shown in Listing 6.13, sets the prediction time to the next 15 minutes starting with the last known value from the respective smart meter. The `derivative` function in Line 8 calculates the difference between the previous and the next predicted value in the given minute granularity, i. e., the watt hours per minute. As the prediction function is linear, the value will be equal for each predicted minute.

```
1 predTime = PREDICTIONTIME({
2   addtostartvalue = [0, 'MINUTES'],
3   addtoendvalue = [15, 'MINUTEs']
4 },temporalize)
5
6 /// Energy consumption per household per minute in the next 15
      minutes
7 derivative = MAP({
8   expressions = [['derivative(temp_wh, PredictionTimes)','whPerMinute
        '],['id','id']]
9 },predTime)
```

Listing 6.13: Calculating the predicted energy consumption

The last part of the query is shown in Listing 6.14. The map function in Line 3 converts this value to watts. The select operator at the end of the query filters out tuples with a low energy consumption (cf. Line 8).

field	value
watt	GenericTemporalType: 51 = 0.0, 52 = 800.00, 53 = 800.00, 54 = 800.00, 55 = 800.00, 56 = 800.00, 57 = 800.00, 58 = 800.00, 59 = 800.00, 60 = 800.00, 61 = 800.00, 62 = 800.00, 63 = 800.00, 64 = 800.00, 65 = 800.00
id	3
stream_time	[51,|∞|)
prediction_time	MINUTES: [[52, 66)]

Table 6.3: Example result of the energy consumption query. Some rounding errors are fixed for better readability.

```
1 /// Energy consumption per household per minute in the next 15
      minutes
2 watts = MAP({
3   expressions = [['whPerMinute * 60','watt'],['id','id']]
4 },derivative)
5
6 /// Filter out elements with a low energy consumption
7 highConsumption = SELECT({
8   predicate = 'watt > 300'
9 },watts)
```

Listing 6.14: Selecting tuples with higher consumption

An example result of the query is listed in Table 6.3. The “watt” attribute contains the consumption over time in the next 15 minutes. Note, that the first value is 0: the derivative cannot be calculated, because the value before the first value does not exist. The select operator selects for values higher than 300 watts, wherefore the first value at prediction time 51 is not within the result. Hence, the `prediction-times` meta attribute starts at 52. The additional value in the `GenericTemporalType` could be removed with the `trimTemporal` map function but is left here for demonstration.

This scenario demonstrates that the approach of temporal attributes can be applied to other fields than only moving objects. For this scenario, no additional functions or changes to the concept are required. Nevertheless, similar to the scenarios with moving objects, the temporal function to represent the development of attributes should be designed for the respective use case.

6.2.9 Scenario Evaluation Summary

The scenario evaluation presents the practical capabilities of the moving object algebra integration into a DSMS, the element join, and the temporalization of non-temporal

attributes, among others. For spatio-temporal queries on data streams from moving objects, the previously defined generic query structure (cf. Section 5.6) is used in scenarios such as the radius and kNN queries. Other queries, for example the CPA query, demonstrate scenarios from the maritime domain (cf. 6.2.4). The energy consumption scenario in Section 6.2.8 gives an example for a non-spatial scenario in which the temporal attributes can also be applied to solve a problem that would otherwise be hard to tackle.

With these scenarios and the application of the more general query structure for spatio-temporal queries, this part of the evaluation focuses on the first part of the research question (cf. 1.2), which asks for a flexible and generic semantics to define queries on spatio-temporal data streams from moving objects. The scenarios show that the existing, generic capabilities of a DSMS can be applied on the new temporal dimension, i. e., with the prediction time. Using the semantics for the prediction time defined in this thesis, queries can be defined in a very generic way and are not limited to certain use cases. Following the ideas of the moving object algebra, even though differences are present due to the streaming approach, a generic semantics is given. With that, the first part of the research question is answered.

The next section tackles the second part of the research question and goes in-depth on performance measurements and efficiency improvements.

6.3 Performance Evaluation

The scenario evaluation in Section 6.2 demonstrates scenarios where the temporal attributes are used to solve challenges with streaming data. The performance evaluation in this section uses parts of those and other scenarios to evaluate the performance of the prototype implementation. Especially the additional costs due to the use of temporalization and temporal attributes are measured. Additionally, the filter and refine concept from Section 5.5 is evaluated. Latency and data rate (or: throughput) are two of the most important performance indicators in a data stream environment [GHN14, AAB+05, CcC+02]. Latency is especially important in scenarios where immediate results are necessary to react on the results as soon as possible.

6.3.1 Evaluation Environment

The performance evaluation is performed on an Linux server running Ubuntu Server version 18.04.1. The Linux kernel version is 4.15.0-36. The system has 64 GiB of memory. Odysseus is configured to use at maximum 50 GiB. The processor is an “Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz”. The data is read from the internal SSD. Odysseus runs on Java. The Java version on the machine is “Java(TM) SE Runtime Environment (build 1.8.0_181-b13)”. The Odysseus version is a build from December 6th, 2018.

For the generation of the box plots for latency and data rate values, each query is executed ten times. The values, such as the medians, are calculated using all the result elements, reaching from a few hundred to a few thousand elements depending on the query. That way, the impact of outliers due to other system processes can be reduced.

6.3.2 Measuring Performance in Streaming Queries

Odysseus offers built-in mechanisms to measure the data rate and latency. The data rate can be measured with the datarate operator. It should be placed at the beginning of a query right after the access operator. That is because at that point in the query, the data rate of the incoming stream is measured. With a placement within the query, for example, after a select operator, the data rate would be lower, i. e., not the real data rate but the selectivity of the operator would be measured.

The latency on the other hand is typically measured within or at the end of a query. The latency measures how long it takes from a stream element entering the query until a stream element created from this input reaches the latency measurement operator. The time that is needed to read the data from disk and write the result to the disk is not part of the measured latency, because the measurement starts when the stream element is already within the query. That way, the measured value is independent from the disk speed.

In Odysseus, the calcLatency operator is used for measuring the latency. It can be placed at multiple stages in the query to measure how the latency is affected by the different operators. The operator in Odysseus already solves some difficulties when measuring the latency. For example, when having a join operator, the latency of a resulting stream element needs to be created with the younger of the two elements. That is because the performance of the operator has to be measured and not the data distribution of the streams. The data distribution could otherwise affect the latency because waiting times, for example in a join operator, would be counted as part of the latency.

For this evaluation, calcLatency operators are placed at different stages in the query, which are described in the following sections. The placement of the measurement operators is depicted in Figure 6.8. At the end of each stage a calcLatency operator is placed so that the impact on the latency of each stage can be measured. The data rate of the whole query is measured at the beginning of the query with a datarate operator.

For the measurement of these values the previously described radius query is used. Each stage can have multiple parameters and with that different configurations, which may have an impact on the performance of the query. The evaluated parameters are described in the following sections. Additionally, the performance of the kNN query is evaluated.

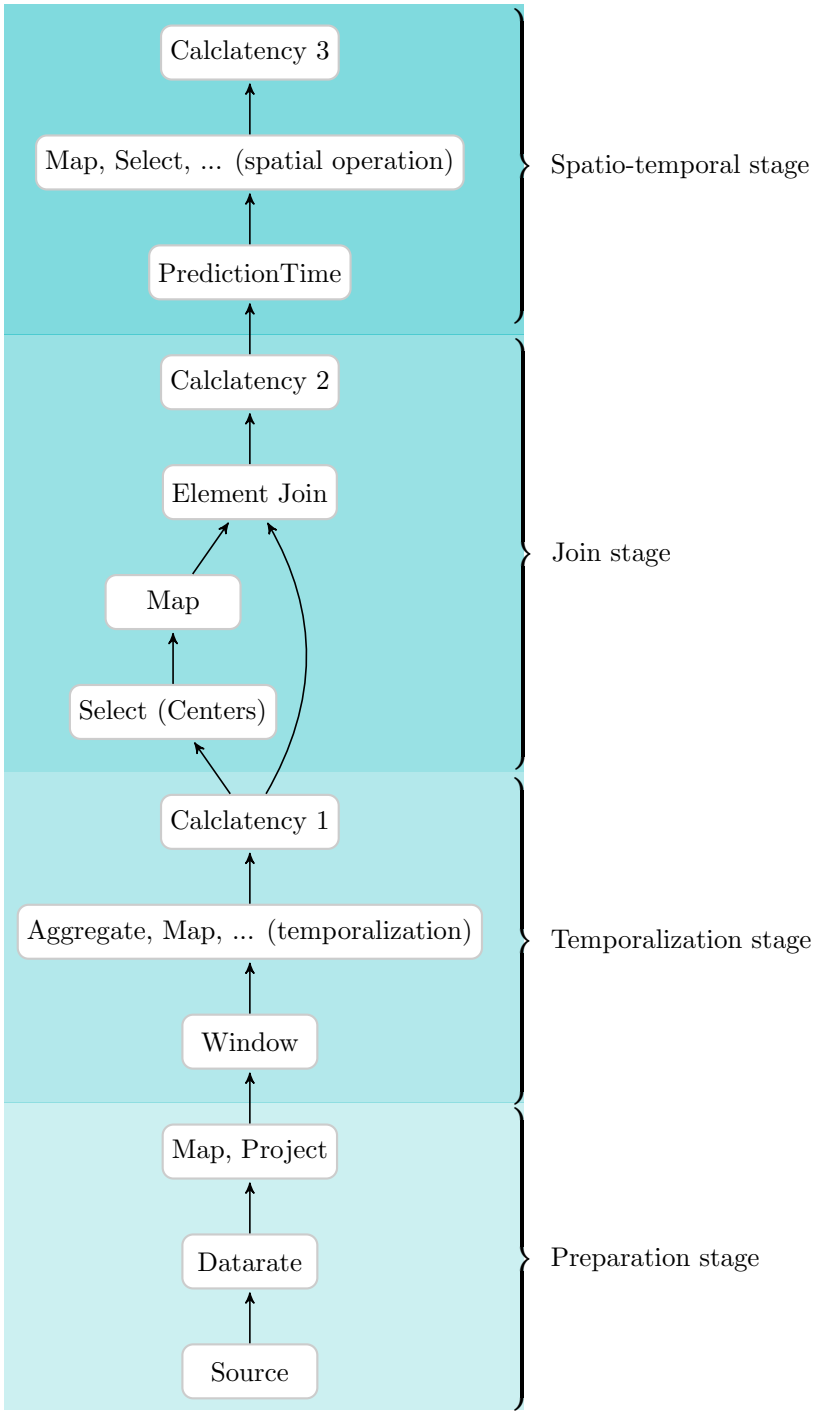


Figure 6.8: Latency and data rate measurements in a typical structure of a spatial query with temporal functions

6.3.3 Costs of the Preparation and Temporalization

The first steps of a query with temporal attributes are often the preparation of the data and the conversion from a non-temporal attribute to a temporal attribute (cf. 5.6). The complexity of this step depends on the process of temporalization. The temporalization method, the window size as well as the complexity of the attribute (e. g., a point or a polygon) are parameters that impact the performance of the temporalization step.

To evaluate the costs of the temporalization, different temporalization methods with different window sizes are compared to each other. The window sizes can be of significance because the aggregation operator, which is used for the temporalization functions, stores the stream elements that are currently in the window. As window sizes, 10, 20 and 30 minutes are used. Hence, all location updates from the vessels within this time period are available to the temporalization functions. For the temporalization step, the functions `ToLinearTemporalPoint` and `ToSplineTemporalPoint` are applied. A part of the query is shown in Listing 6.15.

```
time = TIMEWINDOW({
    size = [60, 'minutes']
  },input)

[...]

/// Temporalize the location attribute
temporalize = AGGREGATION({
          aggregations = [
            ['function' = 'ToSplineTemporalPoint', 'input_attributes'
                 = 'SpatialPoint', 'output_attributes' = '
                temp_SpatialPoint']
          ],
          group_by = ['id'],
          eval_at_outdating = false
  },time)
```

Listing 6.15: Part of the evaluated query with the window and the aggregation operators

Figure 6.9 shows the results for the different queries[3]. As can be seen, the time needed for the temporalization step with these algorithms is very low. With about 0.012 ms and lower in median the temporalization step uses only a very limited portion of the overall latency, as will be discussed in the next sections. The spline algorithm is slightly faster than the linear algorithm. This could be due to a faster implementation in the integrated

[3] The "Without Outliers" at this and the following figures means that the outliers are not printed in the box plot for better readability, not that the outliers are removed from the data. Hence, the data still contains the whole data including the outliers, wherefore the printing of the rest of the plot is not compromised.

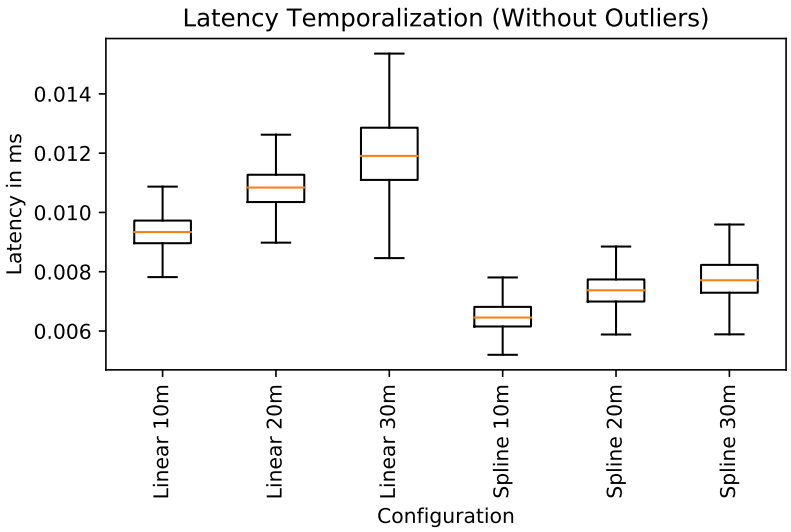


Figure 6.9: Latency of the temporalization queries in ms

spline library compared to the self-written linear algorithm. Nevertheless, the absolute difference is negligible. The latency grows slightly with larger window sizes.

6.3.4 Costs of the Join Stage

For spatio-temporal queries on multiple moving objects, the join stage combines the location information from different moving objects. For this purpose, the previously described element join (cf. 4.6) can be used. In this part of the evaluation, the impact of the join on the performance is measured. For that purpose, the calcLatency operator measures the latency right behind the join operator. The join operator produces, depending on the configuration, a lot of new elements and with that a high load on the following spatio-temporal operations. Odysseus evaluates the stream elements depth-first, wherefore a waiting situation can occur. The join waits for the following operators to finish before it continues to produce new join elements. When that happens, the latency of the join stage goes up without the join being the bottleneck.

To reduce the impact of that effect in the measurements, a (threaded) buffer has been inserted after the latency measurements. The buffer stores the incoming elements in-memory. Therefore, the join can continue to produce new results at full speed without needing to wait for the following operators. Additionally, the buffer allows to run the spatio-temporal stage in multiple threads, potentially increasing the throughput. Depending on the scenario and the size of the buffer, the buffer can become filled and unable to buffer more elements. In that case, the same waiting situation will occur and the

latency of the join will increase. In the standard configuration, a buffer size of 200 000 elements is used.

Listing 6.16 shows an example configuration of a join operator.

```
1 allObjects = TIMEWINDOW({
2   size = [60, 'minutes']
3 },createSpatialObject )
4
5 centerObjects = TIMEWINDOW({
6   size = [1, 'MILLISECONDS']
7 },renameCenter)
8
9 JOIN({
10   predicate = 'id != id_center',
11   elementsizeport0 = 1,
12   elementsizeport1 = 1,
13   group_by_port_0 = ['id_center'],
14   group_by_port_1 = ['id']
15 },centerObjects,allObjects)
```

Listing 6.16: Configuration of an element join which is triggered on elements on the left port

Listing 6.16 shows the definition of the window and join operators. In this example, the join is triggered on each stream element from the left input (`centerObjects`) and joins this new center element with the latest stream element from each other moving object (`allObjects`). The other stream elements are only considered for one hour. If a moving object did not send a location update within this time span, it is no longer considered.

These configurations, i. e., the window sizes and the way the center objects are selected, are the most important parameters for the evaluation of the join stage. 20 elements were randomly chosen from the data set as the center elements. The queries were run with 2, 4, 6, ..., 20 center elements. To keep the number of configurations low, the window size for the center element stream is set to either one time instance (i. e., one millisecond) or to 60 minutes. The former reflects a trigger on center elements only, the latter additionally reflects a trigger on each non-center stream element.

Figure 6.10 shows a box plot of the latencies of all configurations. Two main observations can be made. First, the median latency is significantly lower for queries where all stream elements trigger a join, i. e., with a 60 minutes (60m) window on the non-center input port of the join. The query also has results with a higher latency as indicated by the whiskers. Nevertheless, the median value is lower because of the joins which are triggered by the non-center elements. These produce less join results and therefore have a lower load on the following operators, resulting on lower latencies.

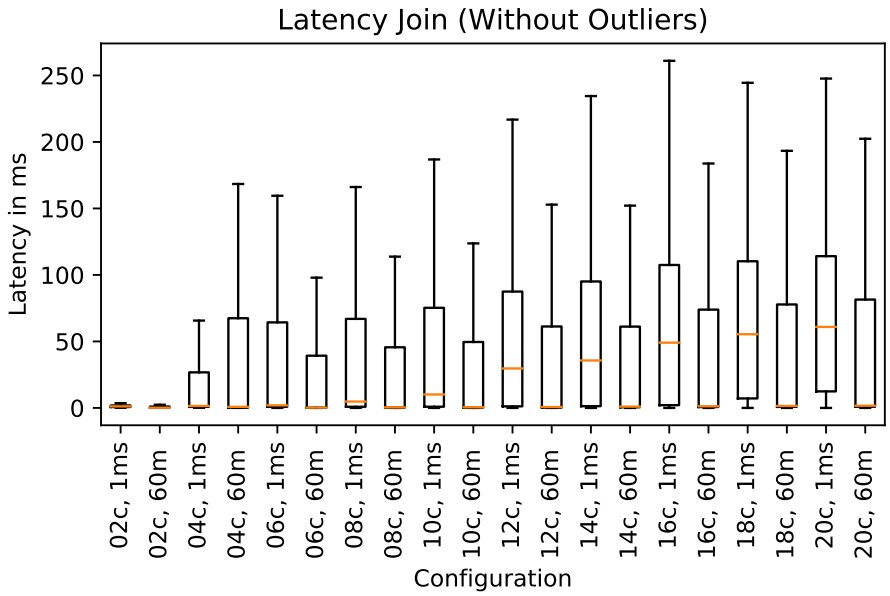


Figure 6.10: The latencies of the join stage with different configurations and a buffer with a size of 200 000 elements.

The second observation is that the median latency increases with the number of center vessels. This is because the number of elements produced by the join increases with the number of center elements. Therefore, the created elements have to wait longer for the next spatio-temporal stage. This congestion leads to higher latencies.

The described behavior of the effects in Figure 6.10 can better be seen in a latency plot over time. Figure 6.11 plots the latency of the result elements over time. The latency of the elements is low as long as the buffer is not filled. Later, the latency has a lot of high peaks. The number of peaks can be reduced or eliminated with a bigger buffer. Figures 6.12 and 6.13 make this effect visible. With the increasing buffer size, the latencies for the different queries are lowered in comparison to Figure 6.10, as the box plot in Figure 6.15 shows. The more elements are produced (on the right of the plot), the more likely that the buffer is too small. The remaining peaks are probably situations where the garbage collector of the Java Virtual Machine (JVM) cleans up the memory.

The characteristics of the peaks can better be seen in a more detailed view. Figure 6.14 shows such a detailed view on a part of the plot from Figure 6.11. Such a peak increases step by step and falls down sharply at its end. This can be explained by the join behavior: when a center element arrives at the join operator, it is joined with all non-center elements (depending on the point in time this can roughly be about a thousand elements). The first joined element can be processed by the next step immediately. The second has to wait a little for the previous element and so on. The last element has to wait the longest. In this plot, only roughly above 150 elements are in the plot because not every

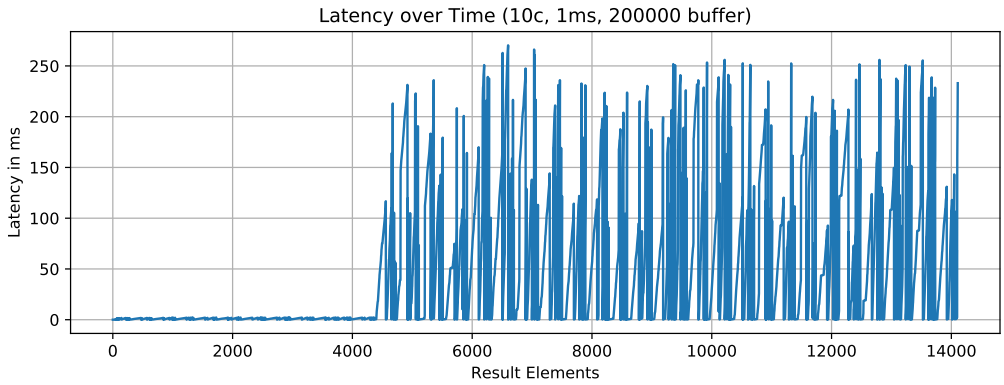


Figure 6.11: The latency of the join stage over time with 10 center elements, a center window of size 1 ms and a buffer with a size of 200 000 elements.

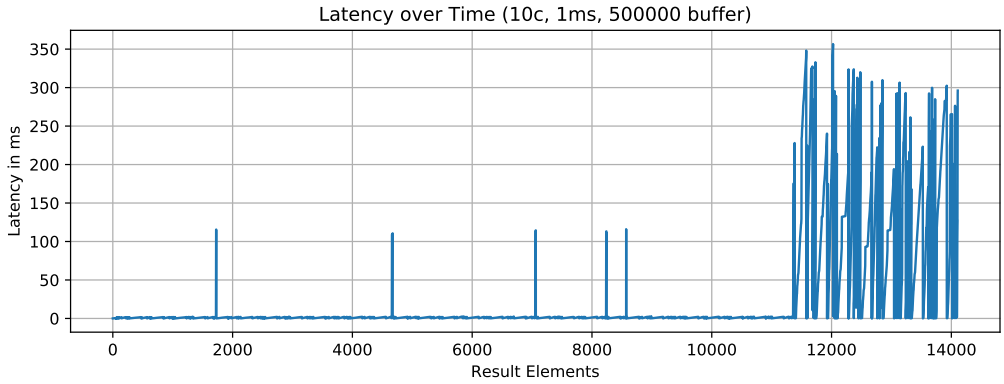


Figure 6.12: The latency of the join stage over time with 10 center elements, a center window of size 1 ms and a buffer with a size of 500 000 elements.

joined element will be in the result set after the spatio-temporal step. Nevertheless, the tendency of the behavior is visible anyway.

Real-Time Data Rate

The described scenarios are a test of the maximum capabilities of the created system. Nevertheless, an important measurement is if the system is capable to run the queries in real-time, i. e., to keep up with the pace of the stream. That is the case if the data rate of the processing measured in this evaluation is higher than the data rate of the original data stream. For this test, no buffer should be used as it would distort the results of this scenario with a limited stream compared to an unbounded stream.

The data set has 81 132 data elements in two hours (cf. 6.1). Thus, the data rate of the live input stream would be roughly $\frac{81132 \text{ tuples}}{2 \cdot 60 \cdot 60 \text{ seconds}} = \frac{81132 \text{ tuples}}{7200 \text{ seconds}} \approx 11.3$ tuples per second.

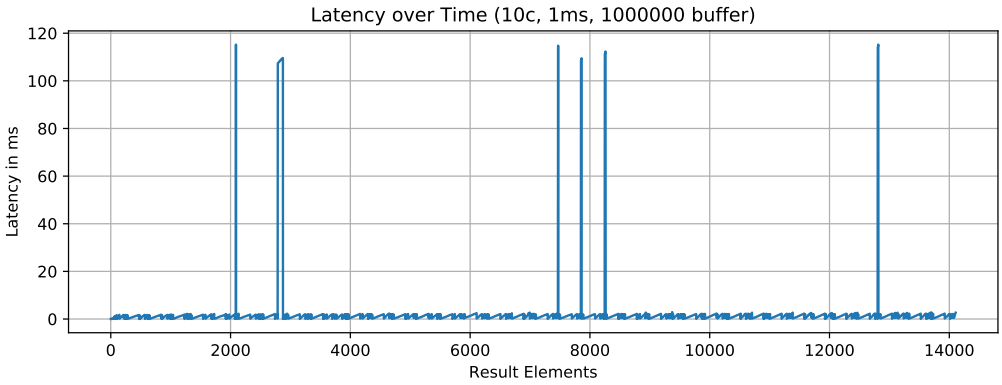


Figure 6.13: The latency of the join stage over time with 10 center elements, a center window of size 1 ms and a buffer with a size of 1 000 000 elements.

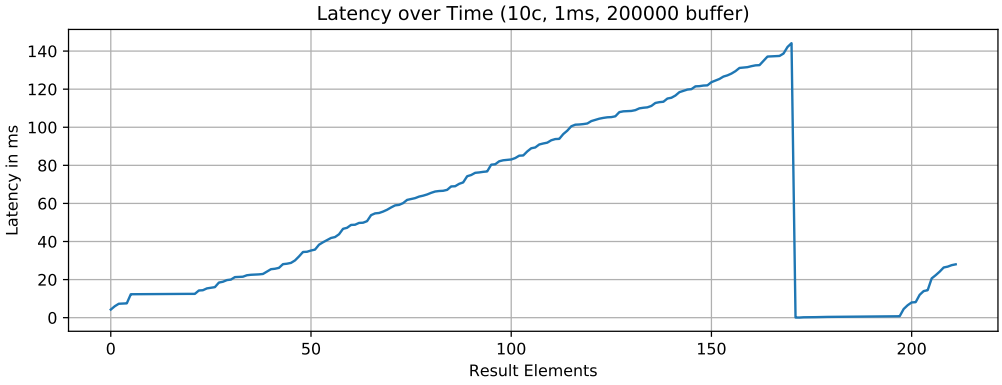


Figure 6.14: Detail view of the latency of the join stage over time with 10 center elements, a center window of size 1 ms and a buffer with a size of 200 000 elements.

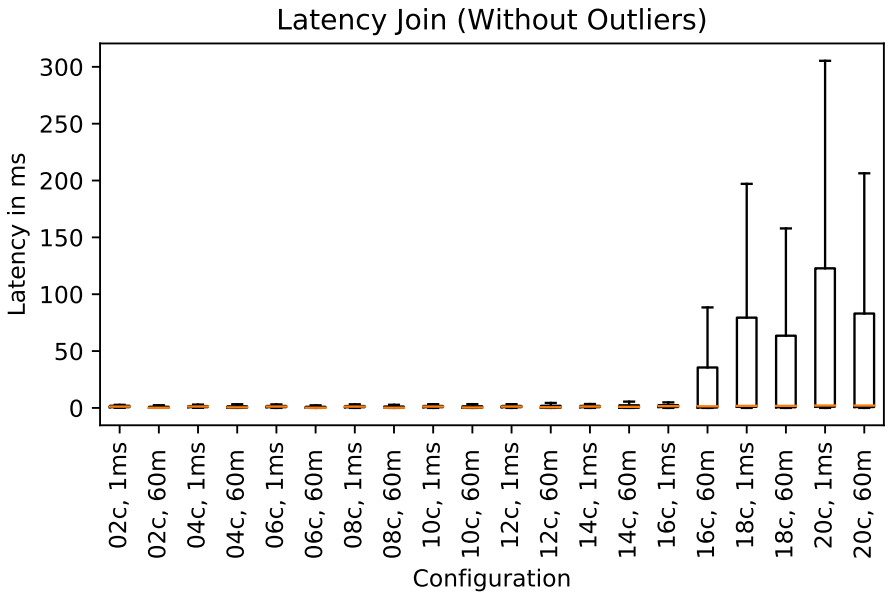


Figure 6.15: The latencies of the join stage with different configurations and a buffer with a size of 1 000 000 elements.

If the evaluated data rate is higher than this value, the system can keep up with the real-time data rate.

Figure 6.16 shows the data rates of different configurations of the query with different numbers of observed center elements, no buffer, ten points in time in the prediction time dimension and a radius of 5000 meters. Note that the measured values are for the whole query, not only for the join stage. As can be seen, the data rate of the queries with a 60 minutes window on the center-port is remarkably lower than the queries with only a one millisecond window. This is because the join produces more results in that configuration, which leads to a higher load on the following operators. Nevertheless, the data rate is in all cases above the average real-time data rate of this specific data set. For example, the lowest value is 238 tuples per second for the “20c, 60m” configuration.

In conclusion, the join stage has a significant impact on the overall query performance. While the join itself does not add too much latency to the query as can be seen in the situations with a non-filled buffer or with an input delay to mimic the real stream data rate, the created load due to the higher amount of stream elements can lead to a congestion in the stream processing pipeline especially because of the following spatio-temporal stage. With a real-time stream (with the chosen original scenario), no congestion occurs. The effect of congestion can be decreased with an in-memory buffer. The next section discusses the last stage of this query.

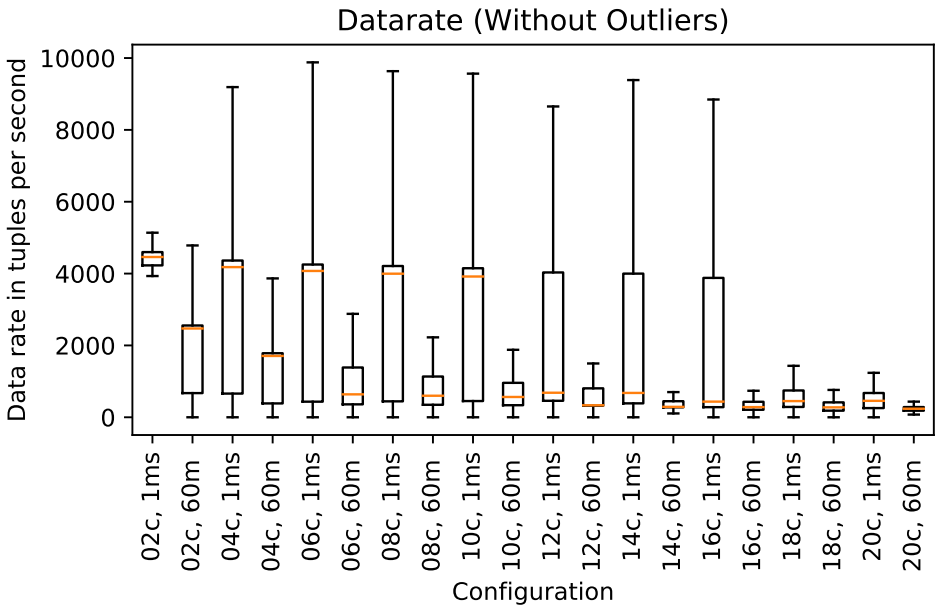


Figure 6.16: Data rate of the radius query with different configurations and without a buffer.

6.3.5 Spatio-Temporal Stage

The last step in the presented generic moving object query structure (cf. 5.6) is the spatio-temporal calculation. As a common spatial or spatio-temporal query, a radius query is evaluated in this section. Again, the join parameters are an important factor for the performance of the query. As they have been evaluated independently, the join parameters are fixed in this query and only the relevant parameters for the radius calculation are altered. The other parameters are set to one center element with a trigger only on new center elements and a buffer of 200 000 elements.

These include mainly the prediction time interval and the granularity of the prediction time. Additionally, the size of the radius can be altered. An example configuration is shown in Listing 6.17. Here, the prediction time includes ten points in time, hence, for each incoming stream element ten distances have to be calculated and the distance predicate in the select operator needs to be evaluated ten times.

```
/// Set the prediction time
predTime = PREDICTIONTIME({
  addToStartValue = [0, 'SECONDS'],
  addToEndValue = [10, 'SECONDS'],
  PredictionBaseTimeUnit = 'SECONDS'
},input)

/// Calculate exact distance
calcDistance = MAP({
  expressions = [['OrthodromicDistance(center_temp_SpatialPoint,
      temp_SpatialPoint)','tdistance']],
  keepInput = true
},predTime)

/// Select vessels within the radius
distanceSelect = SELECT({
  predicate = 'tdistance < 1000'
},calcDistance)
```

Listing 6.17: Spatio-temporal stage of a radius query

For the evaluation of the impact of the prediction time, the radius has been fixed to 5 000 meters and the granularity to one minute. Then, the prediction time has been set to the next 10, 20 and 30 minutes.

The radius is evaluated independently to reduce the number of different configurations. The prediction time is set to the next ten minutes with a one minute granularity. For the radius, 5 000, 10 000, 15 000 and 20 000 meters are used. With a growing radius, more vessels are expected to be within the result set, which may has an impact on the performance.

Figure 6.17 depicts the latencies of the different radius configurations. The main takeaways from this plot are that (1) the latency of the whole radius query for this configuration stays below 200 ms and that (2) the size of the radius does not affect the latency of the query in a way that it would be significant for the use case. The reason for this lies in the way the query works: it calculates the distance between the center element and all joined elements (i. e., all other elements) and then does a comparison of the predefined radius and the calculated distances. Hence, the number of distance calculations and comparisons is equal for different sizes of the radius. As the selection of the elements in the radius is the last step of the query, the number of selected elements does not influence the performance of following operators. Therefore, no performance differences can be measured.

As expected, the latency behavior over time, which has already been described in the join stage (cf. 6.3.4), can be observed in this stage, too. Figure 6.18 shows the effect with the result elements of a full query. In fact, it is the same waiting time as in the previous

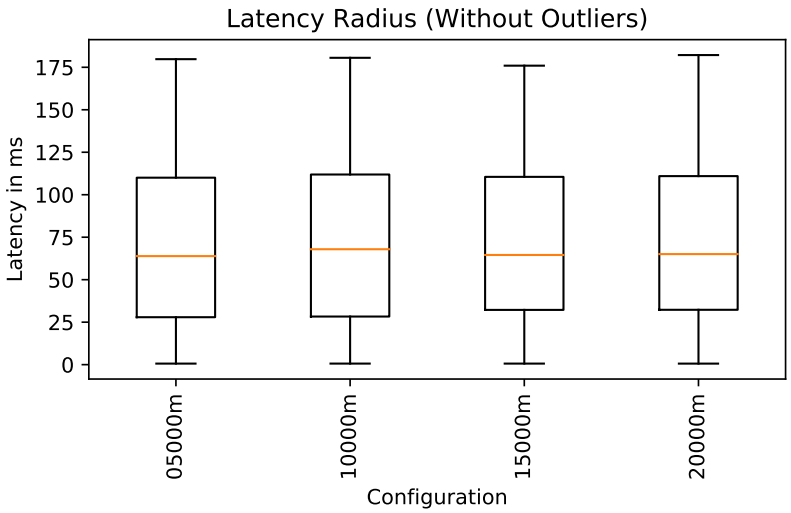


Figure 6.17: Latencies of the full radius query with different radii and a buffer of 200 000 elements, one center element and a center window of 1 ms.

stage. When a center element is joined, the latency grows for each joined element and then shrinks abruptly when all elements are joined for that specific center element.

Prediction Time

Next, the impact of the prediction time interval on the query latency is measured. The results can be seen in Figure 6.19. The left three boxes show the latencies for a query with one center element, the right three boxes for 20 center elements.

The median latency for the first three queries grows from about 61 ms for 10 prediction minutes over 123 ms to 206 ms. As can be seen, the latency grows roughly linear with the number of points in time to predict, which is expected. Internally, the spatial calculation is done multiple times, one time for each prediction minute. Hence, when creating a query, the prediction time should be carefully chosen for the right balance between time span and granularity versus the computational costs.

The right three boxes show that the latency is only slightly negatively affected when increasing the number of center elements. For the scenario with 20 center elements, the median latency is nearly identical. The reason is that the computational costs for a single stream element flowing through the query graph is not affected by the number of center elements. In each case, a stream element that is a center element needs to go through the same steps, among others the join stage and the spatio-temporal stage. Hence, the time a single element needs to be processed does not change, there are just more stream elements that need to be processed (one after each other). Nevertheless, this only affects the data rate, as has been shown in Figure 6.16.

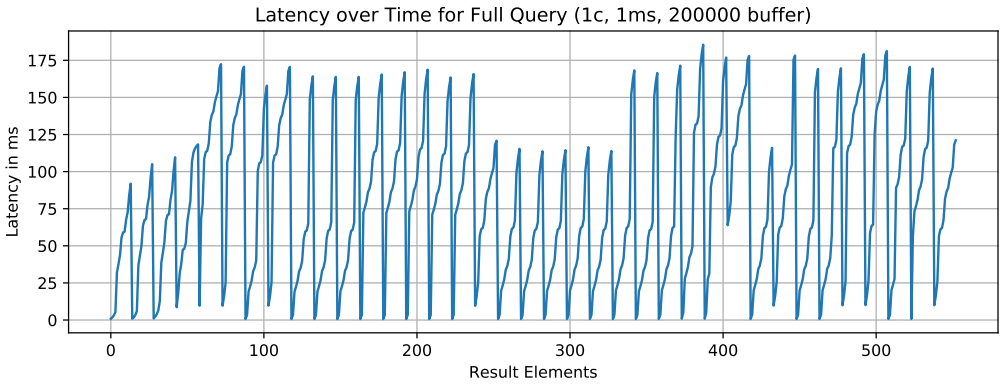


Figure 6.18: Latency of one full radius query over time with a radius of 5 000 meters, a buffer of 200 000 elements, one center element and a center window of 1 ms.

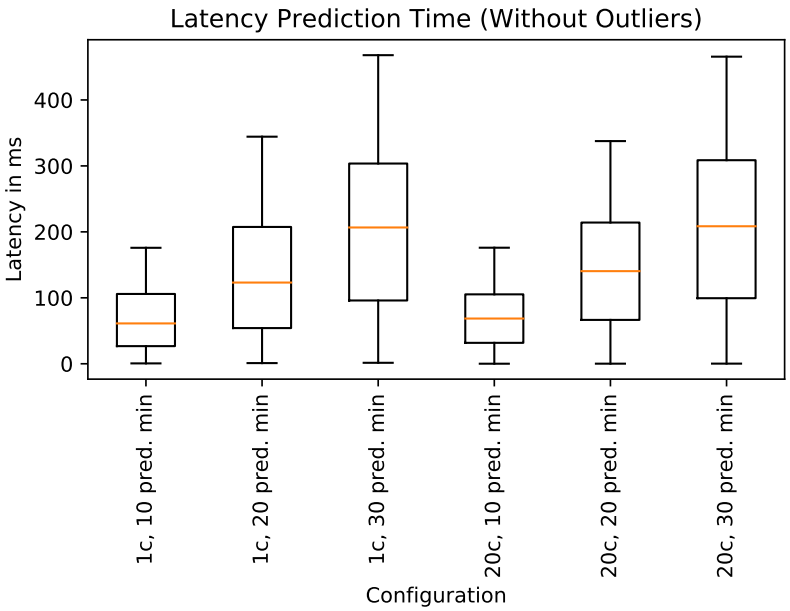


Figure 6.19: Latency of the full radius query with a radius of 5 000 meters, no buffer and a center window of 1 ms.

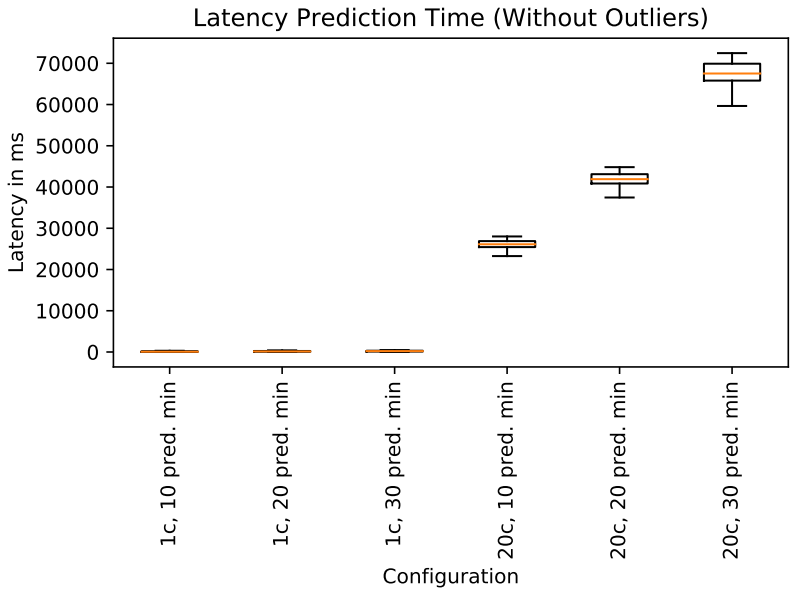


Figure 6.20: Latency of the full radius query over time with a radius of 5 000 meters, a buffer of 200 000 elements and a center window of 1 ms.

Buffer

The previous measurement has been done without any buffer. Figure 6.20 shows the latencies when using a buffer with a size of 200 000 elements. As can be seen, the latencies, especially of the queries with 20 center elements, grow significantly to median values of 26, 42 and 67 seconds. That is because of the waiting time within the buffer. The join can work faster, but the elements have to wait longer before the spatio-temporal processing is finished.

In conclusion, the query latency is significantly influenced by the configuration, especially in the join operator and with the prediction time configuration. When measuring the performance of the stages independently, e. g., by using a buffer, it is clearly visible that the spatio-temporal stage is the bottleneck in this query. The median latency of the whole query is in all tested configurations below one second. Nevertheless, the evaluation also shows that a performance improvement in the spatio-temporal stage is desirable. For that purpose, the filter and refine concept is evaluated later in this chapter. Before going on with the optimized radius query, another spatio-temporal query is evaluated in the next section.

6.3.6 kNN Query

Next to the radius query, the kNN query is also a common spatial query. It seems to be conceptually similar to the radius query, as it also calculates objects which are close to another object, but works differently. Instead of a select operator for the spatial stage, an aggregation with a top k function is used. Therefore, it is another interesting query to evaluate.

Listing 6.18 shows the spatial stage of the kNN query. Again, the prediction time is defined in Line 1 and then the distance between the joined objects is calculated. In Line 10, a time window is set, because the element join beforehand has removed the window information (cf. 4.7.1). The window here includes all the elements that have been joined with the center element with the last trigger, because they have the same start timestamp (the start timestamp from the trigger). Hence, the aggregation uses all the latest elements for the `TopK` function. A tumbling window is used to create exactly one result per center element.

For this query, again, the prediction time has an impact on the performance. Additionally, the "k" parameter of the `TopK` function (here set with `'TOP_K' = '2'`) and the prediction function can impact the performance. Because the impact of the prediction time has already been evaluated for the radius query, the varying parameter here is only the "k". The query is analyzed with "k" set to 10, 20, 30 and 40. For the prediction function, the `ToLinearTemporalPoint` temporalization function is used, which creates a linear function. The query is executed with one center element which sends its location 37 times during the two hours of the data set. The query uses a buffer with a size of 200 000 elements, to have it comparable to the radius query above.

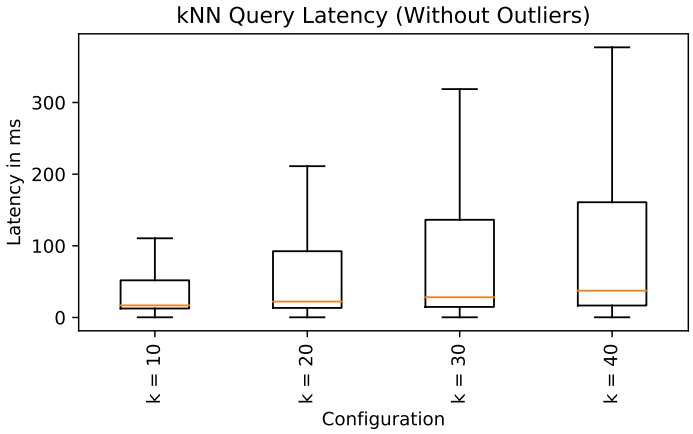


Figure 6.21: The latency distribution for different values for k in a kNN query.

```
predTime = PREDICTIONTIME({
  addToStartValue = [0, 'MINUTES'], addToEndValue = [10, 'MINUTES'],
       predictionBaseTimeUnit = 'MINUTES'
},input)

calcDistance = MAP({
  expressions = [['OrthodromicDistance(center_temp_SpatialPoint,
      temp_SpatialPoint)','tdistance']],
  keepinput = true
},predTime)

timeWin = TIMEWINDOW({size = [1, 'SECONDS'], advance = [1, 'SECONDS
    ']},calcDistance)

AGGREGATION({
  aggregations = [['FUNCTION' = 'TopK', 'TOP_K' = '10', '
      INPUT_ATTRIBUTES' = ['id', 'tdistance'], 'SCORING_ATTRIBUTES' =
       'tdistance', 'UNIQUE_ATTR' = 'id']],
  eval_at_outdating = false
},timeWin)
```

Listing 6.18: Spatio-temporal stage of a kNN query

Figures 6.21 and 6.22 show the evaluation results. As can be seen, the median latency grows moderately with the *k* parameter from about 17 to about 37 ms for k set to ten and 40, respectively. The data rate decreases slightly from about 4450 down to about 4200 tuples per second with an increasing *k*. Hence, the data rate is way above the original data rate of the data set.

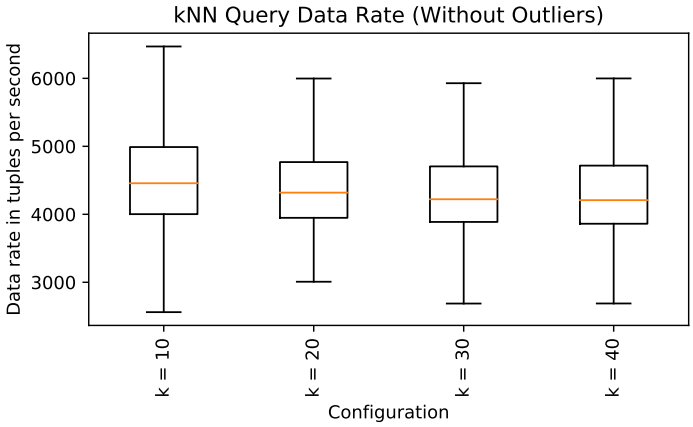


Figure 6.22: The data rate distribution for different values for k in a kNN query.

6.4 Filter Approaches in Moving Object Queries

The previously described and evaluated queries use the straightforward structure for spatio-temporal queries from Section 5.6. To increase query performance, Section 5.5 describes how a filter and refine approach can be applied to streaming spatio-temporal queries. In this section, this approach is evaluated to measure the impact on the query performance as well as on the completeness of the result. The approach is evaluated on the radius and the kNN query.

6.4.1 Optimized Radius Query

To evaluate the performance improvement of the filtering approaches, the latencies and data rates of the same spatio-temporal query are compared while applying different filtering techniques. The following filtering techniques and configurations were used:

- **no filter:** Normal radius query (cf. 6.2.1) with a radius set to 5 000 m
- **approx. dist.:** Approximate distance (cf. 5.5.1) with maximum speed of vessels set to 6 meters per second
- **single rect.:** Single rectangle (cf. 5.5.2) with additional radius set to 3 000 m
- **multi rect.:** Multi rectangle (cf. 5.5.3) with additional rectangles around the radius set to three times 1 000 m

Figure 6.23 depicts the latencies of the optimized queries with an inserted filter step compared to the non-optimized queries. As can be seen, the latencies can be reduced significantly from around 70 ms to around 5 to 8 ms. The effect is also visible in the data rate measurements. Especially for more demanding queries with more center elements,

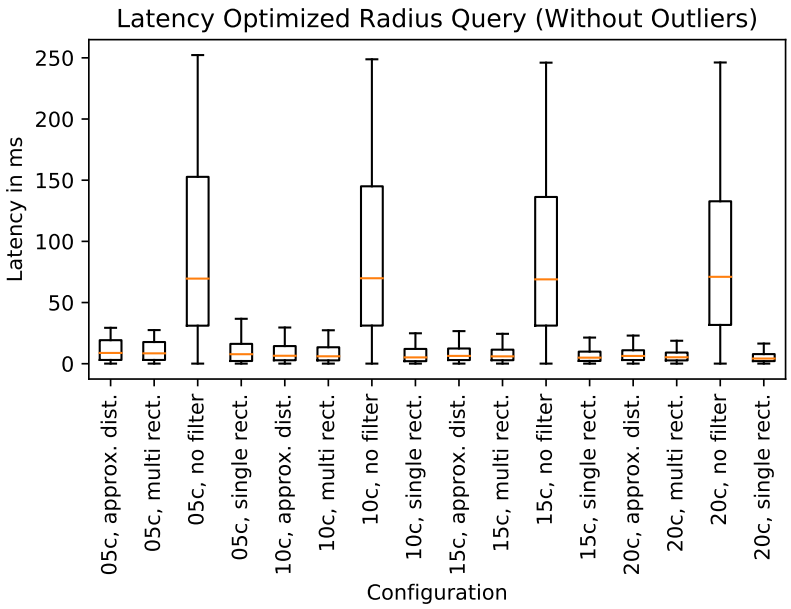


Figure 6.23: Latencies of optimized and not optimized radius queries without any buffer and with ten points in time in the prediction time interval. 05c indicates five center elements, 10c ten center elements and so on.

such as 15 and 20, the data rate of the non-optimized queries are way lower than the data rates of the optimized queries.

Queries can be more demanding with more center elements or with more points in time to predict, among other configurations. In these examples, only ten minutes with a granularity of one minute (i. e., ten points in time) were predicted. When predicting more points in time, the filtering techniques will improve the performance advantage compared to a non-optimized query even more.

When taking a closer look at the optimized queries on Figure 6.25, two trends are visible. First, the median latency is reduced with a growing number of center elements. That can be explained with the decreasing data rate (see Figure 6.24) for more center elements. When the elements enter the stream in a lower pace, they have a lower waiting time after the join before they can enter the spatio-temporal stage.

The second trend is that the single rectangle performs slightly better than the other filter approaches in every query while the approximate distance performs worst. The reason is that the rectangle only needs to be calculated once for each new center element and then only a spatial `within` needs to be calculated. The approximate distance needs to be calculated for each element after the join, which is more demanding than the `within` calculation. The multi rectangle may be better in filtering out more elements,

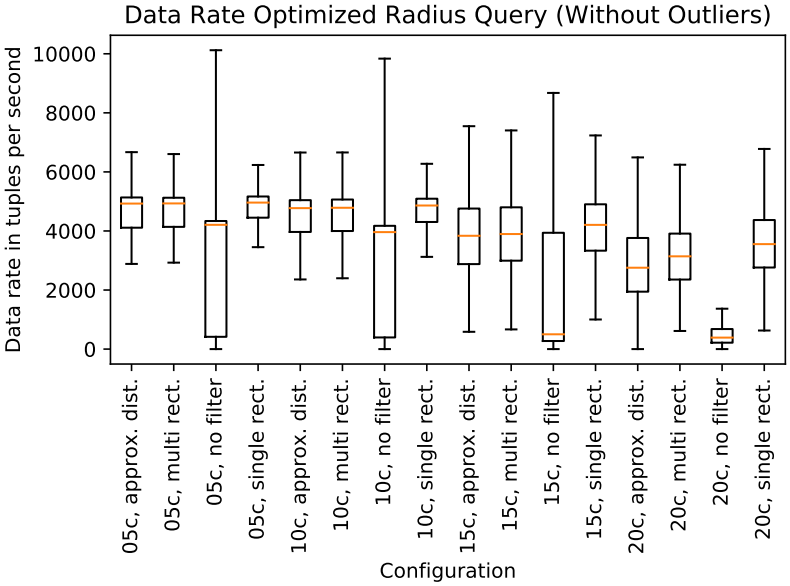


Figure 6.24: Data rates of optimized and not optimized radius queries without any buffer and with ten points in time in the prediction time interval. 05c indicates five center elements, 10c ten center elements and so on [BG19].

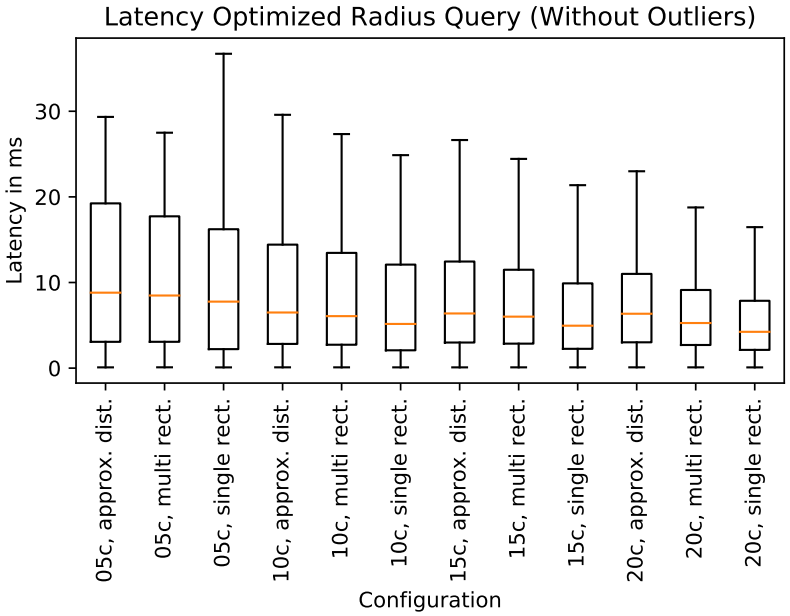


Figure 6.25: Latencies of optimized radius queries without any buffer and with ten points in time in the prediction time interval. 05c indicates five center elements, 10c ten center elements and so on.

but this effect seems to be not worth the higher complexity of calculating more rectangles and later on deciding which rectangle to use.

Even though the performance can be improved significantly, the presented filtering techniques have a downside: it is possible that they miss results. With the evaluated configurations, the numbers of missed results are rather low. The results are best visible for the scenario with 20 center elements, because this scenario has the most result elements. The original result set from the radius query without any filter contains 43 488 elements. The approximate distance filter misses one result, the rest of the result set is equal to the original one.

The single rectangle and multi-rectangle methods both miss 28 results, i. e., about 0.064%. As expected, they miss exactly the same number, because the three time 1000 m from the multi rectangle query are equal to the one time 3000 m from the single rectangle query. Additionally to the low number of missed elements it has to be said that those elements are left out that are far away from the respective center element and therefore have a possibly lower prediction quality anyway. Nevertheless, if it needs to be guaranteed that all possible results are in the result set, the filtering approaches cannot be applied.

In conclusion, the filter approaches for the radius query are a possible and effective method to improve the performance of the query. They miss only very few possible results but improve the latency and data rate significantly, especially for more demanding queries. The best configurations for the filtering approaches depend on the use case. The values used here were only a good guess based on the nature of the data (e. g., the speed at which vessels move). With a more sophisticated parameter tuning that balances between result completeness and performance improvement, the results may be even more improved.

6.4.2 Optimized kNN Query

The previously evaluated filter techniques are made for the radius query. They filter out elements based on the distance to a center object. The kNN also uses the distance of objects to calculate the nearest neighbors, but the actual distance of the elements is not known in advance. Therefore, the presented filtering techniques may perform worse for the kNN query. This is evaluated with a kNN query with ten center elements and k set to ten. In other words, the query searches for ten different elements for the nearest ten elements. To have comparable results for the different queries, a result element is created at maximum once every minute for each center element.

Figures 6.26 and 6.27 show the performance comparison between a non-optimized query and the previously described three optimized queries. For all of the filtering techniques, some settings are necessary. For the evaluation, a radius of 40 km for the approximate distance query and the single rectangle has been used. For the multi-rectangle

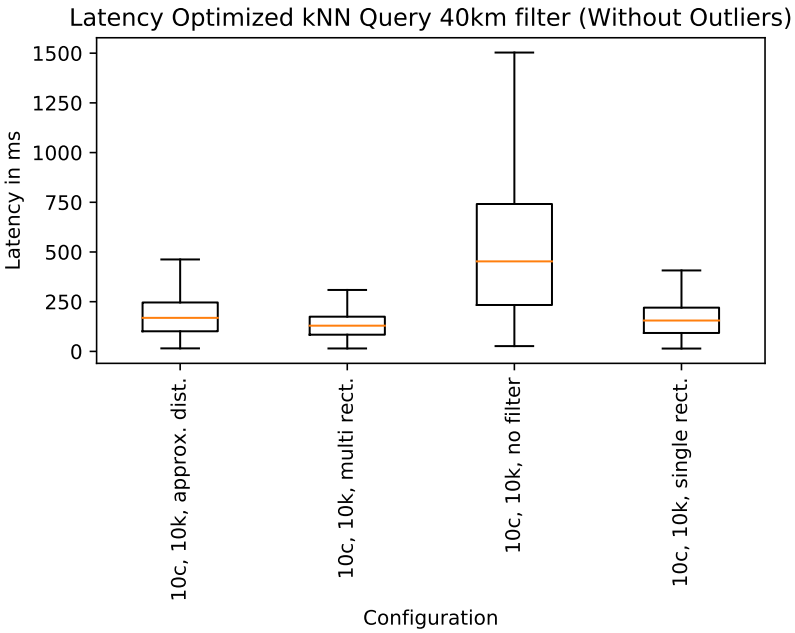


Figure 6.26: Box plot of the latency of optimized and not optimized kNN queries.

method, three times 13.333 km has been used. Here, a major weakness of the approaches for the kNN query is visible: the density of the objects has to be guessed and the assumed density is identical for each part of the data. Unfortunately, that is not the case in the data set for the evaluation: on the ocean, the vessel density is lower than in a harbor.

As expected, the performance of the query can be improved with the introduction of a filtering step. The latency shrinks from a median of about 480 ms to about 160 ms for the optimized queries. It can be seen that the multi-rectangle method performs best with a median of around 130 ms compared to the around 150 ms of the single rectangle and about 160 ms of the approximate distance. The data rate is also improved from about 3350 tuples per second to about 4770 (multi rectangle), 4540 (single rectangle) and 4460 (approximate distance) tuples per second (all median values).

The evaluation shows a significant improvement of the performance of the queries when using the filtering approaches. The multi rectangle approach performs best both in the latency as well as in the data rate evaluation. Nevertheless, an important question is if the filtering step removes relevant results. In this important measurement, the multi rectangle performs significantly worse compared to the two other methods. While the approximate distance with 94% and the single rectangle with 95% equal results compared to the non-filtered version come very close to a perfect result, the multi rectangle only creates 12% equal results and misses 62 of the 509 result elements completely. The two other methods create all results and only miss a few of the closest elements that are very (more than 40 km) far away from the center element. For example, instead of

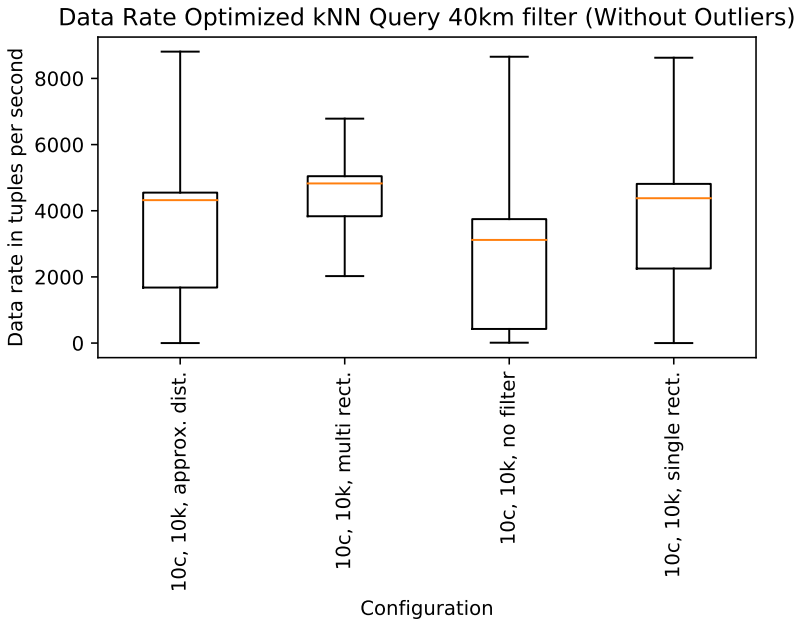


Figure 6.27: Box plot of the data rate of optimized and not optimized kNN queries.

finding the required ten elements, only eight are found in some cases, wherefore for this part of the result the answer would be considered as 80% equal.

The poor performance of the multi-rectangle method can be easily explained. As the temporal distance of the elements is small (probably only a few minutes in most cases), in all cases only the smallest rectangle is used. With its 13 km, a lot of results are missed. The approach of the multi rectangle filter does not work with the way the kNN query is implemented here. Other ways of implementing this approach could be possible, for example by learning the density of the vessels in certain areas and choosing the rectangle based on that information instead on the possible temporal distance.

In the case that the filter radius is set to 10 km instead of 40, the poor performance can also be seen for the other methods. The amount of correct results shrinks down to about 10 to 12%. Nevertheless, the performance improvement stays similar to the improvement of the 40 km filter. Hence, parameter tuning could improve the results even further and are specific for the query and the properties of the data. All in all it can be seen that the filtering techniques also work for kNN queries but in the case of the multi rectangle would need to be adapted.

6.4.3 Key Takeaways

The filter approaches evaluated in this section can be applied for radius and kNN queries on data streams and are capable to improve the performance significantly, both the latency and the data rate. Nevertheless, the approaches depend on a few parameters which have to be tuned for the specific query and the properties of the data. When having unsuitable parameter settings, the filters can lead to a significant loss of relevant results, as can be seen in the kNN query (cf. 6.4.2). The filter approaches work best for radius queries but can also be applied on kNN queries. For good results with the multi rectangle approach in the kNN query, the approach would need to be adapted for the way the kNN query is implemented.

6.5 Summary

The evaluation of the integration of temporal attributes into Odysseus is split into two parts to cover the two parts of the research question (cf. 1.2). The first half of the research question asks for a flexible and generic semantics to express queries on moving object data streams. The first part of the evaluation in this chapter in Section 6.2 uses common generic query types such as the radius and the kNN query to show that these can be expressed with the implemented system. Scenarios in the maritime domain, based on AIS data described in Section 6.1, illustrate how the temporal algebra can be used for practical use cases. Another scenario without moving objects from the energy domain in Section 6.2.8 illustrates that the system is generic enough to be applied to a wide range of use cases.

The second half of the research question asks for the performance of the queries. Therefore, the second part of the evaluation in Section 6.3 takes a closer look at the performance indicators of the queries and with that of the implemented temporal algebra. The costs of the different typical steps in a spatio-temporal streaming query are measured and important parameters for the performance of the queries are determined. Especially the join and prediction time settings have a significant impact on the query performance. The evaluated queries all performed significantly faster than real-time and should therefore be applicable for real use cases with the data and scenarios used for the evaluation.

In the concept, filtering techniques are presented to improve the performance of spatio-temporal queries even further (cf. 5.5). These filtering techniques are evaluated in Section 6.4. The evaluation shows that the performance of the queries can be significantly improved. Based on the parameters, the results of the queries with filters are identical or very close to the queries without a filter. Nevertheless, when using unsuitable parameters, the query can miss a significant part of the original results. The presented techniques work best for radius queries but can also be applied for kNN queries. Never-

theless, especially for the multi rectangle approach, further adaptation for the kNN query would be necessary.

All in all, the evaluation shows that the derived concepts from the moving object algebra and the concept of another temporal dimension are applicable in the data streaming context, with moving objects and even in scenarios without a spatial context. The new capabilities in a DSMS allow for more flexible queries especially when dealing with temporal differences in the data. The impact on the performance depends on the parameters but can be significantly improved with the presented filtering techniques.

The goal of this chapter is to evaluate if and how the concept and implementation of this work answers the research question from Chapter 1. Based on the results of this evaluation, the research question can be considered as answered. This work shows how queries on spatio-temporal data streams from moving objects can be expressed and executed using a DSMS as the foundation and by extending it with a temporal algebra and spatio-temporal capabilities. The evaluation shows that not only can the queries be executed, but also that they can be processed efficiently. That goal is achieved by a new approach to use a join operator with element windows and additionally by adding a filter step to spatio-temporal queries.

7 Conclusion and Future Work

This chapter concludes the thesis about spatio-temporal queries on data streams from moving objects and starts with a summary of the essential parts of the thesis in Section 7.1. The following Section 7.2 reviews and evaluates the results considering the goals of this work and summarizes the contributions. Section 7.3 concludes this thesis with a look on future work that can be done based on this thesis.

7.1 Summary

The motivation behind this work is the usage of spatio-temporal data streams for scenarios with multiple moving objects such as vessels in a densely crowded area. Here, near real-time queries are useful to automatically detect moving objects which are or will be (too) close to each other, which head to forbidden areas or in general to get a better overview of a situation due to better information processing. Running queries on spatio-temporal data streams from moving objects is a necessary capability for these goals. DSMS are a good starting point to reach this goal, as they are made to process data streams. Nevertheless, they lack necessary capabilities for the desired goal, for example, handling temporal gaps between the data stream elements from different moving objects. Based on these goals and challenges, the following research question emerged:

How can queries on spatio-temporal data streams from moving objects be expressed flexibly and with generic semantics and processed efficiently?

This research question has been analyzed in this thesis. The foundation for the analysis are DSMS using the interval approach. The thesis develops an extension for such a DSMS based on the ideas from the moving object algebra. In the following, the most important parts of the thesis are summarized.

Representation of Moving Objects in Data Streams

In a data stream with stream elements, each stream element can have multiple attributes. The attributes can hold data, which in turn can have certain types, for example a string. Unfortunately, DSMS typically cannot represent moving spatial objects, e. g. a moving point. Chapter 3 lays out the foundations to integrate this capability into DSMS.

Instead of adding certain special types for moving points, regions and so on, the concept is kept more generic by adding the possibility to make any attribute a temporal attribute. This way, not only moving points, i. e., points that change over time, but also moving integers, etc. are possible. This idea is based on the concept from the moving object algebra for spatial databases by [GBE+00]. To state more clearly that this concept does not only work for spatial attributes, these attributes are called *temporal* in this thesis, e. g., a temporal integer or a temporal point.

Having the attribute types available is not enough to use them for query processing. The behavior of the operations on those attributes also needs to be defined. Section 3.3 describes the logical integration into a DSMS with time-interval approach. This leads to a bitemporal stream [Bol11]: one time interval from the interval approach and one time interval from the temporal attribute types. One goal of the research question is the flexible and generic semantics, which is achieved with this approach.

An important aspect for the integration is that the existing operations of a DSMS should still be applicable on temporal attributes. If they would need to be reimplemented, many advantages of using a DSMS would be lost. This has been achieved with wrappers around the existing operations, so that the existing logic did not need to be changed and the code changes could be kept small.

Aspects of Physical Integration

Based on the definitions of the logical algebra in Chapter 3, Chapter 4 describes the physical integration with aspects that are closer to the real implementation.

The temporalization is an important first step in a spatio-temporal query. It converts a non-temporal attribute into a temporal attribute with a temporalization function. This function can often be an aggregation function and is then executed by the aggregation operator. Again, existing capabilities from the DSMS have been reused to have a seamless integration. Closely related to this step is the definition and behavior of the time interval for the temporal attributes. Next to the existing stream time interval, the prediction time is introduced in Section 4.3. It defines the time in which the temporal attributes can be used. The behavior of the prediction time is an important aspect of the newly introduced temporal attributes. The section describes how the different operations on the data stream, such as a select operation, affect the prediction time.

Just as non-temporal attributes, temporal attributes can occur in expressions. Expressions with temporal attributes are evaluated for all points in time from in the prediction time interval. The process of translating the expression to a temporal expression is called "lifting", i. e., the resulting expression is called a lifted expression. The aspects of lifted expressions are detailed in Section 4.4, followed by Section 4.5 which explains the temporal trust value that can be used to estimate the trustworthiness of a predicted value.

The previous sections discuss the foundations of temporal attributes and lifted expressions in general. Sections 4.6 and 4.7 show a challenge that occurs in queries with multiple moving objects and develop a generic solution for this challenge. First, in Section 4.6, a query plan for a spatio-temporal radius query is developed. A main downside of this query is a blocking behavior that has its origin in the join behavior and the necessary element windows before it. The blocking behavior is not acceptable for a query that should produce results as soon as new location updates arrive at the system. Therefore, Section 4.7 develops a solution for non-blocking queries by introducing the element join operator, a join with an integrated element window. By integrating the element window

into the join operator itself instead of having the element window in a window operator before the join, the operator does not have to wait for the next element before processing the current element. The details of time interval handling, the algorithm as well as an optimized version of the element join operator are explained in this section. Even though being a more generic solution for stream processing, the element join is a crucial part of the spatio-temporal moving object queries.

The final section in Chapter 4 continues the path to more specific optimizations for moving object queries. In Section 4.8, a spatio-temporal filter concept is developed for a filter and refine step in spatio-temporal queries on moving objects. A challenge for this concept is that moving objects might move over time, but predicting the movement could be computationally expensive and should be avoided. The filter concepts allow a certain approximation and with that a certain inaccuracy with the risk to loose some results for the upside of improving the query performance. The concepts try to estimate the possible movement of the moving objects to filter out those which are too far away to be in the result set. The filter approaches are mainly focused on radius queries.

In conclusion, Chapter 4 develops the physical foundations for moving object queries on data streams. The following chapter uses these foundations to implement them into a system.

Architecture and Implementation

The physical integration represents the theoretical background for the implementation in Chapter 5. The developed approach is implemented into the open source DSMS Odysseus to show the feasibility of using the physical concept with a real DSMS and with real data streams. An important aspect of the implementation is the ability to reuse existing concepts and capabilities of the extended system to benefit from the previous work that has been done for the chosen system.

A main extension to Odysseus is the temporal algebra, consisting of the temporal metadata (the prediction time), temporal attributes and the translation process from non-temporal operators to temporal operators, all described in Section 5.2. Most of these extensions are integrated in a way so that the user can create queries in Odysseus with only a few changes to benefit from the additional temporal operations. For example, to process map operations in the prediction time dimension on temporal attributes, the user can apply existing map functions on those attributes simply as if they would be applied to non-temporal attributes. Odysseus translates those parts of a query into temporal operations in the background and produces temporal attributes as the result. The same is true for other operators such as the select operator and the aggregation operator. When implementing the translation process, the existing operators are reused, i. e., the standard operators do not need to be reimplemented, which was a main goal of the approach.

The element join, the join operator that includes an internal element window, is also implemented into Odysseus. The implementation is described in Section 5.3, followed by a brief description of the already existing spatial operations in Odysseus.

The previous chapter introduced approaches to do a filter step in spatio-temporal queries on moving objects. The three approaches are mainly based on standard stream processing operations. Nevertheless, a few implementations ease the filtering description and process for the query and are described in Section 5.5. Error estimates for the approximate distance are discussed as well as query definitions for the different filter approaches.

Finally, Section 5.6 develops a query structure that can be applied for different spatio-temporal queries on moving object data streams. Based on this query structure and the implemented extensions to Odysseus, the following chapter can evaluate the approach according to the research question.

Evaluation

The evaluation in Chapter 6 evaluates the solution with regard to the research question of this work. The real data that is used for most of the evaluations is described in Section 6.1 and consists of maritime AIS messages from the US coast. To review if and how this work answers the research question, the evaluation looks at the two parts of the research question. First, it is evaluated in Section 6.2 if queries can be “expressed flexibly and with generic semantics”, which is the first requirement from the research question. For the evaluation, data from the maritime domain is used. Different typical queries of this domain, which also represent typical queries on moving objects in general, such as a radius and kNN query, are implemented using the developed system. To show the generic approach of this work, another scenario is presented which does not use any moving object related data, but works in the smart grid domain.

The second important part is the performance evaluation in Section 6.3. It evaluates the second part of the research question: “[...] and processed efficiently”. The performance impact of typical steps in a moving object query is measured with the radius query as a sample query. Latency and data rate values are measured, compared and explained for different configurations to identify the parameters with the main impact on the query performance and to give guidance for users which create queries. A main takeaway from the evaluation is that the tested queries work significantly faster than real-time for the data used in this evaluation. Nevertheless, especially the join configuration which controls which moving objects are observed, and the prediction time configuration which controls how details and how far into the future the queries are predicted, influence the performance of the queries. Another important takeaway is that the spatio-temporal stage of the queries is a main bottleneck, wherefore the filter approaches are likely to improve the queries performance.

These filters are evaluated in Section 6.4. Not only the performance improvement is an interesting measurement, but also how many potential results are lost. A main result of this evaluation is that all three approaches can significantly reduce the latency of the queries, e. g., for an optimized radius query from 70 ms (not optimized) down to 5 to 8 ms for the optimized version. All the filter approaches only miss a very low percentage of the possible results (e. g., 0.065%). Performance-wise, the very simple single-rectangle method performs best, but only slightly better than the other approaches. The evaluation shows that the same methods can also be applied to the kNN query, even if they have slightly more missed results. The multi rectangle approach is not recommendable for the kNN query in its current form. Another results is that for all methods in all queries the parameters have to be tuned to gain good results. All in all, the filter step and the developed approaches work for the presented queries on moving object data streams.

7.2 Contribution

Performing queries on spatio-temporal data streams from moving objects in a flexible manner with high performance, especially low latency, is a non-trivial task. Previous work does not offer a satisfying solution, even though systems for stream processing exist. Therefore, this work offers a solution for this research gap by developing an extension for DSMS that, in a very generic way, allows to perform spatio-temporal queries with another temporal dimension and with multiple moving objects. The developed system is generic enough to be applicable in different scenarios, with different data streams and even in scenarios without a spatial context. This is achieved by integrating a second temporal dimension into the stream processing, inspired by the moving object algebra by [GBE+00].

The contribution of this work is the integration of the aforementioned concepts into DSMS which are build on top of the interval approach by [KS09]. Additionally, a new concept to handle element windows in a join operator with the interval approach has been developed to tackle blocking behavior that occurs in queries with multiple moving objects. Using these new DSMS capabilities, the work designs a generic approach for queries with multiple moving objects and uses this kind of template for common moving objects queries such as radius, kNN and Closest Point of Approach (CPA). Finally, this thesis develops approaches to integrate a filter step for performance improvements in streaming spatio-temporal queries.

Building on top of an existing DSMS and emphasizing reusability of existing capabilities of the system has been proved to be very beneficial. The vast amount of existing operators in the DSMS can now be used with more temporal flexibility and gives the user a great flexibility when designing and implementing queries on data streams in general. The evaluation shows that the queries are not only very flexible, but also perform well, depending on the use case. Especially when applying a filter step before the spatio-

temporal stage of a query, the performance can be greatly improved while maintaining most if not all of the results, depending on the filter approach and the chosen parameters.

Nevertheless, the approach also shows its limits. The new temporal algebra, even though integrated into the system in a way that it can be used in a seamless way together with the existing operators, adds a certain amount of complexity to the query creation process. The query creator has a lot of parameters than can be tuned, which can improve the query significantly, but can also be confusing if someone is not familiar with the system.

The granularity and range of the prediction can decrease the performance significantly. Even though the presented filters can improve the performance, they share the downside that they can miss potential results. Additionally, the thesis does not cover the precision of the prediction itself, but assumes that a method for creating a prediction function from previous data exists for the respective domain and use case. Creating a new function that converts historic movement data to a prediction function is not trivial. In scenarios where the given two functions (linear and spline) are not applicable, the work to create a new function needs to be taken into account.

In conclusion, the work shows a very flexible system to define and execute continuous queries on data streams from moving objects. The system is build on top of an existing DSMS and therefore can reuse most of the available capabilities. Due to this approach, instead of a specialized system for just this use case, the functionality of the created system is extensive and can be extended with ease due to the modular architecture of Odysseus. For example, additionally to the core extensions of this work, a new temporal trust value has been added to the system. Nevertheless, further improvement is possible in future work.

7.3 Future Work

The concepts of this work and the implementation in Odysseus offer a system for streaming moving object queries. Even though the implemented system is capable to be used for certain scenarios, this work also shows interesting research gaps that can be tackled in future work. Some of those shall be briefly discussed in the following.

Temporalization Functions

The temporalization functions introduced in this work are rather simple: they do a linear prediction or use a spline function. These functions are more or less domain agnostic, but could be too simple or inaccurate for some scenarios or domains. Hence, developing better prediction functions and a system for measuring the prediction accuracy could improve the applicability of this system for more scenarios.

In the scenarios shown, historic data from the objects that need to be predicted is available. This could be used for to train better prediction functions. In this field, ma-

chine learning could be an interesting starting point. Due to the nature of the moving object queries, it is typically known where the objects move to. The real location can be compared to the predicted location to improve the prediction function. Additionally, the streaming approach with windows already offers the tools necessary to deal with concept drifts and other streaming related challenges.

Additionally, to create a new prediction function, the source code of Odysseus needs to be extended. Allowing to create and change the temporalization function without the need to change the source code, e. g., with a Domain Specific Language (DSL), could improve the usability of the system for more use cases.

Network-Restricted Scenarios

This thesis focuses on scenarios with moving objects whose movement is not restricted to a certain network, but which can move more or less freely in space. For example, trains are network restricted moving objects, while vessels have more freedom in their movement. While the basic concept of this thesis also works with restricted movement, some concepts can be adapted and studied for the context of network-restricted movement. For example, the prediction functions can improve their prediction quality if they are aware of the network and filter algorithms have to take into account the distance within the network, not the air-line distance.

kNN Filter Algorithm

The presented filter algorithms for the filter step are optimized for radius queries. Even though they can be applied to kNN queries and likely to more spatio-temporal queries, an adaption especially for the multi-rectangle method for kNN queries would be an improvement. The other methods, i. e., the approximate distance as well as the single rectangular method, could also be improved to work better with kNN queries. For example, the algorithm could estimate the density of moving objects in different areas and adjust the extra radius for each area individually.

Automatic Parameter Tuning for Filter Algorithms

The filter algorithms currently have some parameters such as maximum speed of the moving objects that need to be adjusted for the use case. Nevertheless, it should be possible to optimize these settings automatically. For example, the maximum or average speed of moving objects could be learned, even individually for each moving object. The number of rectangles for the multi rectangle algorithm could be optimized using benchmark queries to balance accuracy (not loosing potential results) and performance.

Scalability

Good performance of the created system is a main goal of the research question. This goal is tackled with query design, a special join implementation as well as a filter step for spatio-temporal operations. Another topic for good performance in this field is the scalability of a system. For example, having a radius query for a huge area with a lot of moving objects, is it possible to run this query in a distributed manner on multiple nodes? One basic approach could be to split the query at the join operator, i. e., to have a certain amount of center elements for each node. This should be a comparably simple solution that reduces the bottleneck of the spatio-temporal stage by distributing it to multiple nodes. While it should be possible to manually define a query that way, a dynamic load balancer and query distributor in a cluster of computer nodes would be an interesting research question.

Analytical and Heuristic Expression Evaluation

The current implementation of the evaluation of temporal expressions calculates the non-temporal expression for each point in time in the prediction time interval. This is a very generic solution but can reduce the performance of the evaluation for larger prediction time intervals. Another approach would be to keep functions over time instead of evaluated values for each point in time and solve the functions analytically. For example, if an expression "$x+5$" is applied on the temporal attribute "x", instead of calculating each value for each point in time, the function is stored. When an expression "$x + 5 > 10$" is applied, the function is analytically solved for the intervals for which the expression in true of false. This way, the performance can be improved. This approach is similar to [Bol11].

Another option is to only solve the expressions with a heuristic. Instead of calculating all results, only some results are calculated for which the heuristic assumes that they are relevant or representative. The other values in-between could, again, be interpolated.

Dynamic Query Adjustment

Another more generic topic for data stream processing is the dynamic adjustment of queries while they are running. In the scenarios of this thesis especially the selection of center elements is likely to change. A solution to do this on the fly while the query is running would be an interesting extension to query processing in general. This is not a simple task because of the states of the operators. Simply stopping the running query, changing the parameters of the operators and starting the query again would remove the current states of the operators. Hence, a more sophisticated solution is desirable.

Integration into Query Languages

Currently, the implementation of this work relies on PQL as the query language, which implies some manual work for the user. For some scenarios, an integration into other query languages such as CQL could ease the process of query development and reduce the chance of errors made by the user. For example, the manual manipulation of the prediction time gives a lot of freedom and possibilities, but can also change the semantics of the results in an undesired way. Hence, integrating the prediction time definition in a query language such as CQL, where the possibilities for the user are more limited, could be beneficial for many use cases.

Scenarios with the Prototype

The number of scenarios for which the current prototype has been used is limited to the moving object domain, and, more specific, to the maritime domain. Even though it has been shown that the prototype and general approach of this work can be applied for non-spatial domains, the number of scenarios could be extended. Using the current prototype for different scenarios, especially for real-world problems, could further evaluate what can be done with the current system and which parts should be extended. The moving object domain currently has lots of other use cases due to the ubiquitous availability of location data and internet connections from mobile devices such as connected vehicles (cars, buses, trains, smart bikes, etc.), pedestrians with their smartphones and so on. Upcoming services such as ride sharing as well as existing services that are more and more connected, for example in the logistics domain, demand new solutions for spatio-temporal query processing.

Appendix

A Error of Equirectangular Distance Calculation

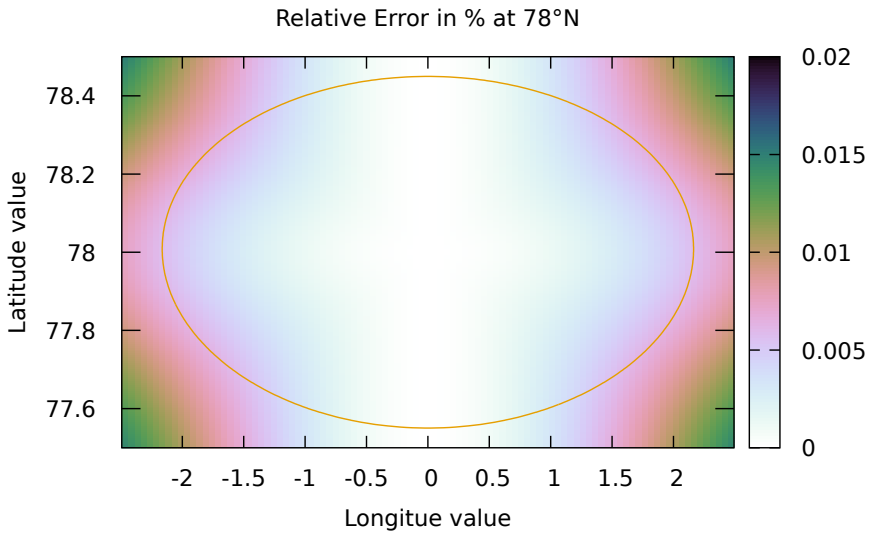


Figure 7.1: Relative error of approximate distance calculation compared to distance calculation with the haversine formula. This is at the height of Spitsbergen. The circle depicts a 50 km distance. Figure is reproduced and slightly modified from [Sal14].

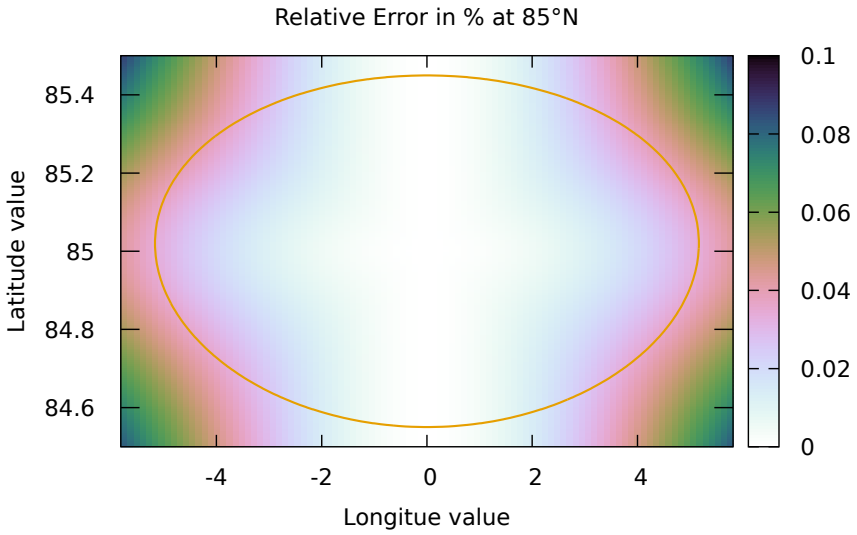


Figure 7.2: Relative error of approximate distance calculation compared to distance calculation with the haversine formula. The circle depicts a 50 km distance. Figure is reproduced and slightly modified from [Sal14].

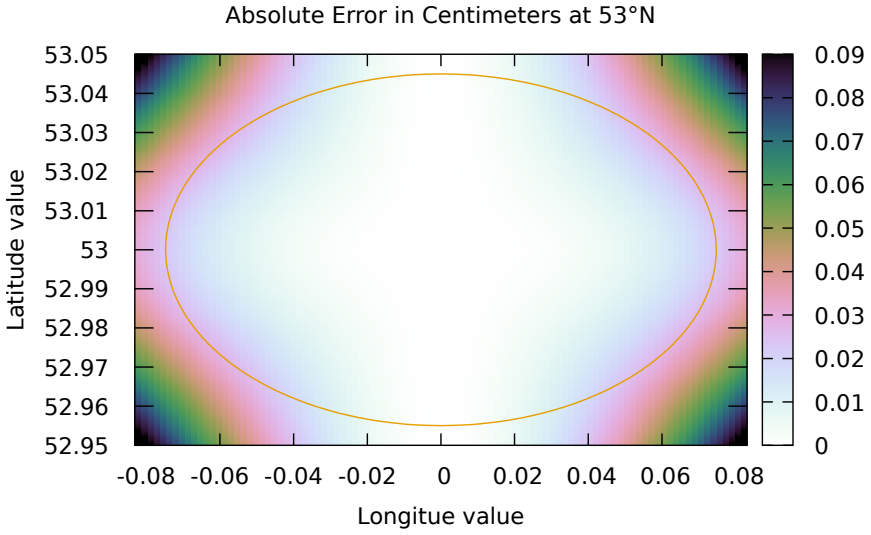


Figure 7.3: Absolute error in meters of approximate distance calculation compared to distance calculation with the haversine formula. Circle marks 5 km radius. Figure is reproduced and slightly modified from [Sal14].

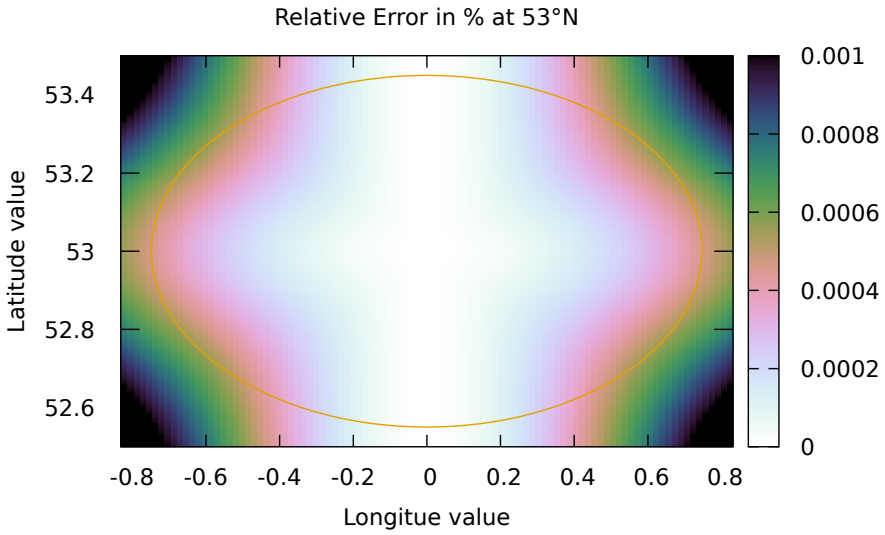


Figure 7.4: Relative error of approximate distance calculation compared to distance calculation with the haversine formula. Circle marks 50 km radius. Figure is reproduced and slightly modified from [Sal14].

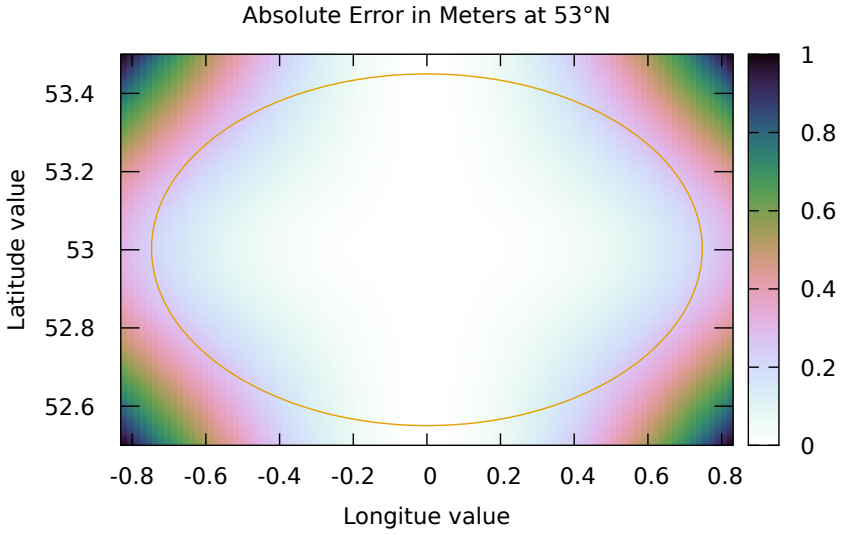


Figure 7.5: Absolute error in meters of approximate distance calculation compared to distance calculation with the haversine formula. Circle marks 50 km radius. Figure is reproduced and slightly modified from [Sal14].

Glossary

Below, essential terms of this theses are summarized and explained. An elaborated explanation can be found in the introducing sections of these terms and in the references which are given there. It is assumed that the reader has background knowledge in the field of computer science, wherefore general terms of this field are not listed. The used symbol ~ in the explanations below refers to the respective term currently explained, the symbol ↑ refers to another term explained in this glossary.

Active data source An ~ sends its data without a previous pull-request, i. e., it pushes the data to one or multiple receivers. Such an active data source could, for example, be a sensor or an ↑AIS sender.

Automatic Identification System (AIS) A system to broadcast information about vessels via radio signals, mainly the identification number and the current location, direction and other information related to the movement of the vessel. The ~ messages are broadcasted via radio signals and can be received with up to around 70 km distance due to the curvature of the earth. Messages from vessels far away from a coast are sometimes received via satellites.

Bitemporal data stream A data stream with two temporal dimensions. One ↑time interval for the window definition in the ↑interval approach and a list of time intervals for the ↑prediction time of the ↑temporal attributes.

Chronon Time between two consecutive ↑points in time. Also see ↑temporal granularity.

Concept Drift A change of the underlying behavior of data that is used to train a model. For example, a model to predict customer behavior for buying sunglasses learned in summer will be less accurate or completely unusable in winter because an underlying concept (the weather) has changed over time [WK96].

Continuous query A ~ is a ↑query over a ↑data stream that is installed once and then runs continuously, producing new results as new ↑data stream elements arrive at the ↑query.

Closest Point of Approach (CPA) The ~ is the location at which two moving objects have the smallest distance between each other. It is similar to the ↑Closest Time of Approach.

Closest Time of Approach (CTA) The ~ is the time at which two moving objects have the smallest distance between each other. It is similar to the ↑Closest Point of Approach.

Data Stream A ~ is a continuously updating source of data from an ↑Active Data Source, which is potentially ↑unbounded and consists of ↑Data Stream Elements.

Data Stream Element A ↑Data Stream consists of single elements of data which arrive at a receiver, e. g., a ↑DSMS, one after each other. Often, the elements consists of a number of attributes following a defined schema. In that case, a ~ can also be called a ↑tuple.

Datum ↑Geodetic datum

Data Stream Management System (DSMS) A ~ is a system to define and execute ↑continuous queries on ↑data streams. It can offer query languages, ↑operators, access methods to different data sources, user management and more.

Expression An ~ is a number of variables and constants concatenated by operators. The variables are attributes in a ↑data stream element. An ~ could, for example, be "`someNumber + 5 < 42`". The result of such an expression would be a Boolean value and can thus also be called a ↑predicate.

Filter and refinement To reduce the costs of a spatial ↑query, a ~ mechanism tries to "reduce the set of objects to be looked at when processing the query" [RSV02]. In one or many ↑filtering steps, the number of spatial objects is reduced, mainly with the use of ↑spatial indices. In the ↑refinement step, the actual, computationally more costly spatial operations of the spatial query are calculated.

Filter step The step before the ↑refinement step in a ↑filter and refinement process. It reduces the number of candidate objects of the result set of a spatial ↑query, often with the help of a ↑spatial index.

Geodesic distance The ~ is "the shortest path between two points on the earth" [Kar13]. When simplified to a sphere, it can, for example, be calculated with the ↑Haversine formula. When treated as an oblate ellipsoid, using the ↑Vincenty's formulae is a way to calculate this distance.

Geodetic datum A coordinate system containing a reference ellipsoid and reference points to map locations to the earth. Also see ↑WGS 84.

Geodetic distance ↑Geodesic distance

GeoJSON A JavaScript Object Notation (JSON) based "format for encoding a variety of geographic data structures"[1].

Global Navigation Satellite System (GNSS) A satellite based system to determine the location on the earth, e. g., via a navigation system. GPS is the most common ~ .

Great-circle distance A distance calculation on a sphere. The ~ is the shortest path between two points on a sphere. See also ↑Geodesic distance.

[1] `http://geojson.org/`

Haversine formula A common formula to calculate the ↑Great-circle distance between two points on the surface of the earth. In contrast to the ↑Vincenty's formulae it is less accurate as it simplifies the earth to a perfect sphere.

Interval approach An approach for ↑windows on data streams. Each ↑data stream element has a time interval, the ↑stream time interval, in the form $[t_S, t_E)$. The start timestamp t_S determines the ↑point in time from when the ↑data stream element is valid. The end timestamp t_E determines the ↑point in time when the ↑data stream element falls out of the window and is no longer used for operations in a ↑query.

JTS Topology Suite (JTS) The ~ is a Java library for creating and manipulating vector geometry [Loc]. It implements a geometry model based on ↑Open Geospatial Consortium standards, geometric functions and spatial structures as well as readers and writers for common data formats such as ↑Well-known text.

Latency The ~ is the time that a ↑DSMS needs to process a ↑data stream element. It is the difference of the ↑point in time at which the element enters the ↑query and the ↑point in time when the element leaves the ↑query.

Location Based Services (LBSs) ~ "can be defined as services that integrate a mobile device's location or position with other information so as to provide added value to a user" [SV04]. ~ often work with location information from ↑Global Navigation Satellite System (GNSS). An example is a smartphone app that provides information about nearby restaurants.

Location update In the context of this work, a ~ is a message (i. e., ↑data stream element) which contains the location of a ↑moving object at a certain ↑point in time.

Moving object A ~ is a spatial object, e. g., a vessel or a pedestrian, that moves in space, in most cases on the surface of the earth. A moving object can often be simplified to a moving point, hence, a spatial object without a shape.

Odysseus An open source data stream management system developed at the University of Oldenburg. It is written in Java, is modular and extensible and offers a set of ↑operators. It implements the ↑interval approach.

Open Geospatial Consortium (OGC) "The OGC (Open Geospatial Consortium) is an international not for profit organization committed to making quality open standards for the global geospatial community" [OGC]. The standards are, for example, used for the ↑JTS Topology Suite.

Operator An ~ is a processing unit in a ↑query which consumes an arbitrary number of input ↑data streams and creates an arbitrary number of output ↑data steams. In-between, the ~ processes the ↑data stream elements.

Orthodromic distance ↑Great-circle distance

Point in time A number $t \in \mathbb{Z}$ that is a timestamp and typically counts up from a certain date in a certain time unit. A common timestamp is the number of milliseconds since January 1^{st} 1970. A ↑time interval typically consists of two timestamps.

Procedural Query Language (PQL) A ↑query language used in the ↑DSMS ↑Odysseus to define ↑continuous queries. In the ~ , the ↑operators with their parameters and the connections between them are defined. Doing this, a ↑query graph is created which can be executed.

Predicate An ↑expression with a Boolean result. A ~ can be used for select and join ↑operators.

Prediction In the context of this work, a ~ is the calculation of a value of a ↑temporal attribute for a certain ↑point in time at which no measured value is known. This can be a prediction into the future (a forecast), a prediction to the current ↑point in time (a nowcast), a prediction to values in-between known values (an interpolation) or a prediction to a ↑point in time before the first known measured value.

Prediction time A ↑point in time in the ↑prediction time interval to which a ↑temporal attribute is predicted to.

Prediction time interval A ↑time interval that includes all ↑points in time to which the ↑temporal attributes in a ↑data stream element can be ↑predicted. As it is possible to predict the attributes to multiple different ↑time intervals, each ↑data stream element in fact has a list of ~ .

Punctuation A ↑data stream element without any content but only a ↑point in time which marks the progress of the time in the ↑data stream.

Query With a ~ , data (e. g., a ↑data stream) can be processes, mainly with the goal to retrieve information. On ↑data streams, ↑continuous queries are used. A ~ can be represented by a ↑query graph.

Query Graph A ~ represents a ↑query in a directed graph (V, E) of ↑operators V. The ↑operators are connected to each other by directed edges E. The ↑data streams flow in the direction of the edges.

Refinement step The ~ in the ↑filter and refinement process uses the set of candidate elements from the previous ↑filtering step and does the exact calculation of the spatial ↑predicate, which is typically more expensive than the filtering.

Route operator The ~ is an operator in ↑Odysseus that routes the incoming stream elements to certain output ports according to a number of ↑predicates. The route operator is similar to parallel select operators. Additionally, the ~ sends a ↑punctuation for each incoming stream element to all output ports to indicate the temporal progress to all subsequent ↑operators in the ↑query graph.

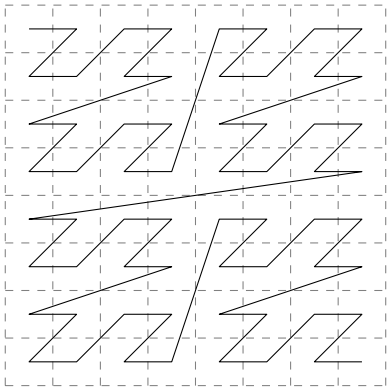


Figure 7.6: An example for a *z*-order curve over a grid of cells.

Space-filling curve A ~ is used to create a proximity-preserving order of cells in a multi-dimensional space. This way, spatial objects such as rectangular cells can be ordered in a one-dimensional space. A popular ~ is the ↑z-ordering curve, which is depicted in Figure 7.6.

Spatial index A ~ is used for the ↑filtering step in the ↑filter and refinement process for spatial ↑queries. An example for a spatial index is an R-tree.

Spatio-temporal data ~ is data with spatial and temporal information. For example, a ↑location update from a ↑moving object contains the location of that object as a point (spatial information) and the ↑point in time at which the location has been measured.

Stateful operator An ↑operator that uses the history of the ↑data stream to process its results. For example, a join or an aggregation in a ↑continuous query are ~ . A ~ should be used after a ↑window operator to avoid an infinite size of the state due to an ↑unbounded ↑data stream.

Stream time interval A ↑time interval at which a ↑data stream element is valid. The ~ can be manipulated by a ↑window operator. It determines, how long a ↑data stream element is used by ↑operators that have a state, i. e., work on the elements in the window (and not only the current element). Such operators are, for example, the join and the aggregation operators.

SweepArea A ~ is an abstract data structure to manage the ↑data stream elements within a ↑window for a ↑stateful operator. It maintains the ↑data stream elements which are in a ↑window and removes the elements which dropped out of a ↑window because they are too old.

Temporal attribute A ~ is an attribute in a ↑data stream element with a ↑temporal type. A ↑data stream element can have multiple ~ and typically has a ↑prediction

time interval. The ~ can, for example, be a temporal integer or a temporal spatial point.

Temporal granularity The amount of time between two consecutive ↑points in time. For example, when having a granularity of one minute, there is one minute between the ↑points in time 1 and 2.

Temporal type A ~ is a type (e. g., an integer or a spatial point) which has a temporal dimension and has for each ↑point in time of the ↑prediction time interval(s) a value of the type.

Throughput A value to benchmark ↑continuous queries on ↑data streams. The ~ describes the number of ↑data stream elements that are processed per second (or a different duration). The higher, the better.

Timestamp ↑Point in time

Time interval A ~ defines a range of ↑points in time. It is typically written as a half-open interval $[t_S, t_E)$, including the start timestamp t_S and excluding the end timestamp t_E.

Trust A value that describes how trustworthy a ↑data stream element is. Especially when ↑predicting a ↑temporal attribute, the ~ can change over time due to a higher or lower accuracy of the ↑prediction.

Tuple A ↑data stream element which attributes follow a certain schema.

Unbounded A ↑data stream is typically ~ , as it is potentially infinite or it is not known if or when the ↑data stream will end. ↑Windows are a concept to work with ~ ↑data streams.

UTM Universal Transverse Mercator (UTM) is a system of map projections. It divides the earth into 60 zones and uses a projection for each zone.

Vessel A more generic term for a ship.

Vincenty's formulae Two methods to calculate the ↑Geodesic distance between two points on the earth. In contrast to the ↑Haversine formula, it is more accurate as it uses an oblate ellipsoid to model the earth.

Vessel Traffic Service (VTS) A ~ monitors and controls maritime traffic, e. g., in ports. Among others, a ~ uses the ↑Automatic Identification System.

Window A concept in ↑Data Stream Management Systems to limit the view on a ↑data stream to a finite set of ↑data stream elements. A window can, for example, be defined over a period of time ("keep the last 30 minutes of the stream") or over a number of elements ("keep the last 100 elements of the stream"). In an ↑continuous query, a window can be defined with a ↑window operator and be implemented with the ↑interval approach.

Window operator The ↑window operator is used to manipulate the ↑stream time interval in the ↑interval approach. The ∼ only manipulates the time interval but does not store any ↑data stream elements.

World Geodetic System 1984 (WGS 84) A very common global spatial reference system for the earth, which defines the flattened ellipsoid that is used as a model of the earth.

Well-known text (WKT) A text format to represent spatial objects. For example, a spatial point can be represented as `POINT (42 21)` with `42` being the x-coordinate and `21` being the y-coordinate.

***z*-order curve** ↑Space-filling curve

Acronyms

AIS	Automatic Identification System
CPA	Closest Point of Approach
CRS	Coordinate Reference System
CTA	Closest Time of Approach
COG	Course Over Ground
CQL	Continuous Query Language
CSV	comma-separated values
DBMS	Data Base Management System
DE-9IM	Dimensionally Extended Nine-Intersection Model
DSL	Domain Specific Language
DSMS	Data Stream Management System
EPSG	European Petroleum Survey Group Geodesy
GCS	Global Coordinate System
GIS	Geographic Information Systems
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
IoT	Internet of Things
JSON	JavaScript Object Notation
JTS	JTS Topology Suite
JVM	Java Virtual Machine
kNN	*k*-nearest neighbors
LBS	Location Based Service
MMSI	Maritime Mobile Service Identity
OGC	Open Geospatial Consortium
PCS	Projected Coordinate System

PQL	Procedural Query Language
SAMS	Safe Automation of Maritime Systems
SFA	Simple Feature Access
UTM	Universal Transverse Mercator
VTS	Vessel Traffic Service
WGS 84	World Geodetic System 1984
WKT	Well-known text

List of Figures

List of Publications

[BG17] Brandt, Tobias ; Grawunder, Marco: Moving Object Stream Processing With Short-Time Prediction. In: *Proceedings of the 8th ACM SIGSPATIAL Workshop on GeoStreaming*. New York, NY, USA : ACM, 2017 (IWGS'17). – ISBN 978–1–4503–5492–9, 49–56

[BG18] Brandt, Tobias ; Grawunder, Marco: GeoStreams: A Survey. In: *ACM Comput. Surv.* 51 (2018), Mai, Nr. 3, 44:1–44:37. http://dx.doi.org/10.1145/3177848. – DOI 10.1145/3177848. – ISSN 0360–0300

[BG19] Brandt, Tobias ; Grawunder, Marco: Spatial Query Processing on AIS Data Streams in Data Stream Management Systems. In: Abramowicz, Witold (Hrsg.) ; Corchuelo, Rafael (Hrsg.): *Business Information Systems Workshops*. Cham : Springer International Publishing, 2019. – ISBN 978–3–030–36691–9, S. 461–472

[Bra17] Brandt, Tobias: Processing Moving Object Data Streams with Data Stream Management Systems. In: *Proceedings of the VLDB 2017 PhD Workshop*, 2017. – ISSN 1613–0073

Bibliography

[AAB+05] Abadi, Daniel J. ; Ahmad, Yanif ; Balazinska, Magdalena ; Çetintemel, Uğur ; Cherniack, Mitch ; Hwang, Jeong Hyon ; Lindner, Wolfgang ; Maskey, Anurag S. ; Rasin, Alexander ; Ryvkina, Esther ; Tatbul, Nesime ; Xing, Ying ; Zdonik, Stan: The design of the Borealis stream processing engine. In: *2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005*, 2005, S. 277–289

[ABC+15] Akidau, Tyler ; Bradshaw, Robert ; Chambers, Craig ; Chernyak, Slava ; Fernández-Moctezuma, Rafael J. ; Lax, Reuven ; McVeety, Sam ; Mills, Daniel ; Perry, Frances ; Schmidt, Eric ; Whittle, Sam: The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. In: *Proc. VLDB Endow.* 8 (2015), August, Nr. 12, 1792–1803. http://dx.doi.org/10.14778/2824032.2824076. – DOI 10.14778/2824032.2824076. – ISSN 2150–8097

[AG05] Almeida, Victor T. ; Güting, Ralf H.: Supporting Uncertainty in Moving Objects in Network Databases. In: *Proceedings of the 13th Annual ACM International Workshop on Geographic Information Systems*. New York, NY, USA : ACM, 2005 (GIS '05). – ISBN 1–59593–146–5, 31–40

[AGB06] Almeida, Victor T. ; Güting, Ralf H. ; Behr, Thomas: Querying Moving Objects in SECONDO. In: *7th International Conference on Mobile Data Management (MDM'06)* Bd. 6, IEEE, May 2006. – ISSN 1551–6245, S. 47

[AGG+12] Appelrath, H.-Jürgen ; Geesen, Dennis ; Grawunder, Marco ; Michelsen, Timo ; Nicklas, Daniela: Odysseus: A Highly Customizable Framework for Creating Efficient Event Stream Management Systems. In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA : ACM, 2012 (DEBS '12). – ISBN 978–1–4503–1315–5, 367–368

[Ala17] Alarabi, Louai: ST-Hadoop: A MapReduce Framework for Big Spatio-temporal Data. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. New York, NY, USA : ACM, 2017 (SIGMOD SRC '17). – ISBN 978–1–4503–4199–8, 40–42

[AR99] Abraham, Tamas ; Roddick, John F.: Survey of Spatio-Temporal Databases. In: *Geoinformatica* 3 (1999), März, Nr. 1, 61–99. http://dx.doi.org/10.1023/A:1009800916313. – DOI 10.1023/A:1009800916313. – ISSN 1384–6175

[BAG+12] Bolles, A. ; Appelrath, H. J. ; Geesen, D. ; Grawunder, M. ; Hannibal, M. ; Jacobi, J. ; Köster, F. ; Nicklas, D.: StreamCars: A new flexible architecture for driver assistance systems. In: *2012 IEEE Intelligent Vehicles Symposium*, 2012. – ISSN 1931–0587, S. 252–257

[BBC+15] Brand, Michael ; Brandt, Tobias ; Cordes, Carsten ; Wilken, Marc ; Michelsen, Timo: Herakles: A system for sensor-based live sport analysis using private peer-to-peer networks. In: Ritter, Norbert (Hrsg.) ; Henrich, Andreas (Hrsg.) ; Lehner, Wolfgang (Hrsg.) ; Thor, Andreas (Hrsg.) ; Friedrich, Steffen (Hrsg.) ; Wingerath, Wolfram (Hrsg.): *Datenbanksysteme für Business, Technologie und Web (BTW 2015) - Workshopband*. Bonn : Gesellschaft für Informatik e.V., 2015, S. 71–80

[BBF+10] Biem, Alain ; Bouillet, Eric ; Feng, Hanhua ; Ranganathan, Anand ; Riabov, Anton ; Verscheure, Olivier ; Koutsopoulos, Haris ; Moran, Carlos: IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2010 (SIGMOD '10). – ISBN 978–1–4503–0032–2, 1093–1104

[BBMS05] Balazinska, Magdalena ; Balakrishnan, Hari ; Madden, Samuel ; Stonebraker, Michael: Fault-tolerance in the Borealis Distributed Stream Processing System. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2005 (SIGMOD '05). – ISBN 1–59593–060–4, 13–24

[BG17] Brandt, Tobias ; Grawunder, Marco: Moving Object Stream Processing With Short-Time Prediction. In: *Proceedings of the 8th ACM SIGSPATIAL Workshop on GeoStreaming*. New York, NY, USA : ACM, 2017 (IWGS'17). – ISBN 978–1–4503–5492–9, 49–56

[BG18] Brandt, Tobias ; Grawunder, Marco: GeoStreams: A Survey. In: *ACM Comput. Surv.* 51 (2018), Mai, Nr. 3, 44:1–44:37. http://dx.doi.org/10.1145/3177848. – DOI 10.1145/3177848. – ISSN 0360–0300

[BG19] Brandt, Tobias ; Grawunder, Marco: Spatial Query Processing on AIS Data Streams in Data Stream Management Systems. In: Abramowicz, Witold (Hrsg.) ; Corchuelo, Rafael (Hrsg.): *Business Information Systems Workshops*. Cham : Springer International Publishing, 2019. – ISBN 978–3–030–36691–9, S. 461–472

[BGA16] Brandt, T. ; Grawunder, M. ; Appelrath, H. J.: Anomaly detection on data streams for machine condition monitoring. In: *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*, 2016, S. 1282–1287

[BGA17] Brand, Michael ; Grawunder, Marco ; Appelrath, H.-Jürgen: A Modular Approach for Non-Distributed Crash Recovery for Streaming Systems. In: Mitschang, Bernhard (Hrsg.) ; Nicklas, Daniela (Hrsg.) ; Leymann, Frank (Hrsg.) ; Schöning, Harald (Hrsg.) ; Herschel, Melanie (Hrsg.) ; Teubner, Jens (Hrsg.) ; Härder, Theo (Hrsg.) ; Kopp, Oliver (Hrsg.) ; Wieland, Matthias (Hrsg.): *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, Gesellschaft für Informatik, Bonn, 2017, S. 309–328

[BGJ+09] Bolles, Andre ; Grawunder, Marco ; Jacobi, Jonas ; Nicklas, Daniela ; Appelrath, Hans-Jürgen: Odysseus: Ein Framework für maßgeschneiderte Datenstrommanagementsysteme. In: *GI Jahrestagung*, 2009, S. 2000–2014

[BKS93] Brinkhoff, T. ; Kriegel, H. P. ; Schneider, R.: Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In: *Proceedings of IEEE 9th International Conference on Data Engineering*, 1993, S. 40–49

[BM72] Bayer, R. ; Mccreight, E. M.: Organization and Maintenance of Large Ordered Indexes. In: *Acta Inf.* 1 (1972), September, Nr. 3, 173–189. http://dx.doi.org/10.1007/BF00288683. – DOI 10.1007/BF00288683. – ISSN 0001–5903

[Bol11] Bolles, André: *Ein datenstrombasiertes Framework zur Objektverfolgung am Beispiel von Fahrerassistenzsystemen*, Carl von Ossietzky University of Oldenburg, Diss., 2011. http://d-nb.info/1014605520

[Bra17] Brandt, Tobias: Processing Moving Object Data Streams with Data Stream Management Systems. In: *Proceedings of the VLDB 2017 PhD Workshop*, 2017. – ISSN 1613–0073

[Bri07] Brinkhoff, Thomas: Geodatenbanken. In: Kudrass, Thomas (Hrsg.): *Taschenbuch Datenbanken*. Leipzig : Carl Hanser Verlag GmbH & Co. KG, 2007, Kapitel 16, S. 496–527

[BSS18] Bakli, Mohamed S. ; Sakr, Mahmoud A. ; Soliman, Taysir Hassan A.: A spatiotemporal algebra in Hadoop for moving objects. In: *Geo-spatial Information Science* 21 (2018), Nr. 2, 102-114. http://dx.doi.org/10.1080/10095020.2017.1413798. – DOI 10.1080/10095020.2017.1413798

[ÇAA+16] Çetintemel, Uğur ; Abadi, Daniel ; Ahmad, Yanif ; Balakrishnan, Hari ; Balazinska, Magdalena ; Cherniack, Mitch ; Hwang, Jeong-Hyon ; Madden, Samuel ; Maskey, Anurag ; Rasin, Alexander ; Ryvkina, Esther ; Stonebraker, Mike ; Tatbul, Nesime ; Xing, Ying ; Zdonik, Stan: The

Aurora and Borealis Stream Processing Engines. In: Garofalakis, Minos (Hrsg.) ; Gehrke, Johannes (Hrsg.) ; Rastogi, Rajeev (Hrsg.): *Data Stream Management: Processing High-Speed Data Streams*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2016. – ISBN 978–3–540–28608–0, 337–359

[CcC+02] Carney, Don ; Çetintemel, Uğur ; Cherniack, Mitch ; Convey, Christian ; Lee, Sangdon ; Seidman, Greg ; Stonebraker, Michael ; Tatbul, Nesime ; Zdonik, Stan: Monitoring Streams: A New Class of Data Management Applications. In: *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB Endowment, 2002 (VLDB '02), 215–226

[CCD+03] Chandrasekaran, Sirish ; Cooper, Owen ; Deshpande, Amol ; Franklin, Michael J. ; Hellerstein, Joseph M. ; Hong, Wei ; Krishnamurthy, Sailesh ; Madden, Samuel R. ; Reiss, Fred ; Shah, Mehul A.: TelegraphCQ: Continuous Dataflow Processing. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2003 (SIGMOD '03). – ISBN 1–58113–634–X, 668–668

[CD16] Christian Denker, Axel H.: MTCAS - An Assistance System for Maritime Collision Avoidance. In: *12th International Symposium on Integrated Ship's Information Systems & Marine Traffic Engineering Conference* DGON, 2016. – In recent years accident statistics have shown a continuous increase in serious and very serious accidents at sea. In the near future, higher traffic density is estimated, which may further contribute to this increase. Within the 3-year Project MTCAS, par

[CHKS04] Cammert, Michael ; Heinz, Christoph ; Krämer, Jürgen ; Seeger, Bernhard: Anfrageverarbeitung auf datenströmen. In: *Datenbank-Spektrum, S* (2004), S. 5–13

[CKE+15a] Carbone, Paris ; Katsifodimos, Asterios ; Ewen, Stephan ; Markl, Volker ; Haridi, Seif ; Tzoumas, Kostas: Apache Flink™: Stream and Batch Processing in a Single Engine. In: *IEEE Data Eng. Bull.* 38 (2015), Nr. 4, S. 28–38

[CKE+15b] Carbone, Paris ; Katsifodimos, Asterios ; Ewen, Stephan ; Markl, Volker ; Haridi, Seif ; Tzoumas, Kostas: Apache Flink™: Stream and Batch Processing in a Single Engine. In: *IEEE Data Eng. Bull.* 38 (2015), Nr. 4, 28–38. `http://sites.computer.org/debull/A15dec/p28.pdf`

[CZC+13] Chen, C. ; Zhang, D. ; Castro, P. S. ; Li, N. ; Sun, L. ; Li, S. ; Wang, Z.: iBOAT: Isolation-Based Online Anomalous Trajectory Detection. In: *IEEE Transactions on Intelligent Transportation Systems* 14 (2013), June,

Nr. 2, S. 806–818. http://dx.doi.org/10.1109/TITS.2013.2238531. – DOI 10.1109/TITS.2013.2238531. – ISSN 1524–9050

[DSTW02] Dittrich, Jens-Peter ; Seeger, Bernhard ; Taylor, David S. ; Widmayer, Peter: Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In: *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB Endowment, 2002 (VLDB '02), 299–310

[EEB+07] Eide, Magnus S. ; Endresen Øyvind ; Brett, Per O. ; Ervik, Jon L. ; Røang, Kjell: Intelligent ship traffic monitoring for oil spill prevention: Risk based decision support building on AIS. In: *Marine Pollution Bulletin* 54 (2007), Nr. 2, 145 - 148. http://dx.doi.org/https://doi.org/10.1016/j.marpolbul.2006.11.004. – DOI https://doi.org/10.1016/j.marpolbul.2006.11.004. – ISSN 0025–326X

[EGSV99] Erwig, Martin ; Güting, Ralf H. ; Schneider, Markus ; Vazirgiannis, Michalis: Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. In: *Geoinformatica* 3 (1999), September, Nr. 3, 269–296. http://dx.doi.org/10.1023/A:1009805532638. – DOI 10.1023/A:1009805532638. – ISSN 1384–6175

[EM13] Eldawy, Ahmed ; Mokbel, Mohamed F.: A Demonstration of SpatialHadoop: An Efficient Mapreduce Framework for Spatial Data. In: *Proc. VLDB Endow.* 6 (2013), August, Nr. 12, 1230–1233. http://dx.doi.org/10.14778/2536274.2536283. – DOI 10.14778/2536274.2536283. – ISSN 2150–8097

[EPS07] EPSG: *WGS 84 – WGS84 - World Geodetic System 1984, used in GPS*. https://epsg.io/4326. Version: 2007

[ESL15] Eom, Sungkwang ; Shin, Sangjin ; Lee, Kyong-Ho: Spatiotemporal query processing for semantic data stream. In: *Semantic Computing (ICSC), 2015 IEEE International Conference on* IEEE, 2015, S. 290–297

[ESRa] ESRI: *Managing spatiotemporal big data stores*. http://enterprise.arcgis.com/de/geoevent/latest/administer/managing-big-data-stores.htm

[ESRb] ESRI: *Working with spatial references*. http://help.arcgis.com/en/sdk/10.0/arcobjects_net/conceptualhelp/index.html#//0001000002mq000000

[FB74] Finkel, R. A. ; Bentley, J. L.: Quad Trees a Data Structure for Retrieval on Composite Keys. In: *Acta Inf.* 4 (1974), März, Nr. 1, 1–9. http://dx.doi.org/10.1007/BF00288933. – DOI 10.1007/BF00288933. – ISSN 0001–5903

[FEHL13] Fox, A. ; Eichelberger, C. ; Hughes, J. ; Lyon, S.: Spatio-temporal indexing in non-relational distributed databases. In: *2013 IEEE International Conference on Big Data*, 2013, S. 291–299

[FXX+13] Feng, Shenzhu ; Xu, Jian ; Xu, Ming ; Zheng, Ning ; Zhang, Xiaofei: EHSTC: An Enhanced Method for Semantic Trajectory Compression. In: *Proceedings of the 4th ACM SIGSPATIAL International Workshop on GeoStreaming*. New York, NY, USA : ACM, 2013 (IWGS '13). – ISBN 978–1–4503–2532–5, 43–49

[GAA+05] Guting, R. H. ; Almeida, V. ; Ansorge, D. ; Behr, T. ; Ding, Z. ; Hose, T. ; Hoffmann, F. ; Spiekermann, M. ; Telle, U.: SECONDO: an extensible DBMS platform for research prototyping and teaching. In: *21st International Conference on Data Engineering (ICDE'05)*, 2005. – ISSN 1063–6382, S. 1115–1116

[Gal16] Galić, Zdravko: *Spatio-Temporal Data Streams*. New York, NY : Springer New York, 2016. http://dx.doi.org/10.1007/978-1-4939-6575-5. http://dx.doi.org/10.1007/978-1-4939-6575-5. – ISBN 978–1–4939–6575–5

[GBE+00] Güting, Ralf H. ; Böhlen, Michael H. ; Erwig, Martin ; Jensen, Christian S. ; Lorentzos, Nikos A. ; Schneider, Markus ; Vazirgiannis, Michalis: A Foundation for Representing and Querying Moving Objects. In: *ACM Trans. Database Syst.* 25 (2000), März, Nr. 1, 1–42. http://dx.doi.org/10.1145/352958.352963. – DOI 10.1145/352958.352963. – ISSN 0362–5915

[GBKM14] Galić, Z. ; Baranović, M. ; Križanović, K. ; Mešković, E.: Geospatial data streams: Formal framework and implementation. In: *Data & Knowledge Engineering* 91 (2014), 1 - 16. http://dx.doi.org/https://doi.org/10.1016/j.datak.2014.02.002. – DOI https://doi.org/10.1016/j.datak.2014.02.002. – ISSN 0169–023X

[Gei13] Geisler, Sandra: Data Stream Management Systems. Version: 2013. http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:0030-drops-42975. In: Kolaitis, Phokion G. (Hrsg.) ; Lenzerini, Maurizio (Hrsg.) ; Schweikardt, Nicole (Hrsg.): *Data Exchange, Integration, and Streams* Bd. 5. Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013. – URN urn:nbn:de:0030–drops–42975. – ISBN 978–3–939897–61–3, 275–304

[GFW+11] Grün, Thomas von d. ; Franke, Norbert ; Wolf, Daniel ; Witt, Nicolas ; Eidloth, Andreas: A Real-Time Tracking System for Football Match and Training Analysis. In: Heuberger, Albert (Hrsg.) ; Elst, Günter (Hrsg.) ;

Hanke, Randolf (Hrsg.): *Microelectronic Systems: Circuits, Systems and Applications*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–23071–4, 199–212

[GHN14] Geesen, Dennis ; Herzog, Otthein ; Nicklas, Jun-Prof Dr D.: *Maschinelles Lernen in Datenstrommanagementsystemen*. OlWIR, Oldenburger Verlag für Wirtschaft, Informatik und Recht, 2014

[GLW08] Gudmundsson, Joachim ; Laube, Patrick ; Wolle, Thomas: Movement Patterns in Spatio-temporal Data. In: *Encyclopedia of GIS*. Boston, MA : Springer US, 2008. – ISBN 978–0–387–35973–1, 726–732

[GMKB12] Galić, Zdravko ; Mešković, Emir ; Križanović, Krešimir ; Baranović, Mirta: OCEANUS: A Spatio-temporal Data Stream System Prototype. In: *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on GeoStreaming*. New York, NY, USA : ACM, 2012 (IWGS '12). – ISBN 978–1–4503–1695–8, 109–115

[Gro] Group, The O.: *4.16 Seconds Since the Epoch*. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16

[GSF11] Gariel, M. ; Srivastava, A. N. ; Feron, E.: Trajectory Clustering and an Application to Airspace Monitoring. In: *IEEE Transactions on Intelligent Transportation Systems* 12 (2011), Dec, Nr. 4, S. 1511–1524. http://dx.doi.org/10.1109/TITS.2011.2160628. – DOI 10.1109/TITS.2011.2160628. – ISSN 1524–9050

[Gut84] Guttman, Antonin: R-trees: A Dynamic Index Structure for Spatial Searching. In: *SIGMOD Rec.* 14 (1984), Juni, Nr. 2, 47–57. http://dx.doi.org/10.1145/971697.602266. – DOI 10.1145/971697.602266. – ISSN 0163–5808

[Gü93] Güting, Ralf H.: Second-order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 1993 (SIGMOD '93). – ISBN 0–89791–592–5, 277–286

[Gü94] Güting, Ralf H.: An Introduction to Spatial Database Systems. In: *The VLDB Journal* 3 (1994), Oktober, Nr. 4, 357–399. http://dl.acm.org/citation.cfm?id=615204.615206. – ISSN 1066–8888

[HAF+03] Hammad, Moustafa A. ; Aref, Walid G. ; Franklin, Michael J. ; Mokbel, Mohammed P. ; Elmagarmid, Ahmed K.: Efficient execution of sliding-window queries over data streams. (2003)

[HHD12] Hoque, M. A. ; Hong, X. ; Dixon, B.: Analysis of mobility patterns for urban taxi cabs. In: *2012 International Conference on Computing, Networking and Communications (ICNC)*, 2012, S. 756–760

[HMPR04] Hevner, Alan R. ; March, Salvatore T. ; Park, Jinsoo ; Ram, Sudha: Design Science in Information Systems Research. In: *MIS quarterly* 28 (2004), Nr. 1, S. 75–105

[HRH+09] Hasan, Khondker S. ; Rahman, Mashiur ; Haque, Abul L. ; Rahman, M A. ; Rahman, Tanzil ; Rasheed, M M.: Cost effective GPS-GPRS based object tracking system. In: *Proceedings of the international multiconference of engineers and computer scientists* Bd. 1, 2009, S. 18–20

[Hub12] Huber, William A.: *How accurate is approximating the Earth as a sphere?* https://gis.stackexchange.com/questions/25494/how-accurate-is-approximating-the-earth-as-a-sphere#25580. Version: 2012

[HXL05] Hu, Haibo ; Xu, Jianliang ; Lee, Dik L.: A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2005 (SIGMOD '05). – ISBN 1–59593–060–4, 479–490

[HZEF16] Hughes, James N. ; Zimmerman, Matthew D. ; Eichelberger, Christopher N. ; Fox, Anthony D.: A Survey of Techniques and Open-source Tools for Processing Streams of Spatio-temporal Events. In: *Proceedings of the 7th ACM SIGSPATIAL International Workshop on GeoStreaming*. New York, NY, USA : ACM, 2016 (IWGS '16). – ISBN 978–1–4503–4579–8, 6:1–6:4

[IH11] Inc., Open Geospatial C. ; Herring, John R.: *OpenGIS ® Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture*. http://www.opengeospatial.org/standards/sfa. Version: 2011

[JDB+98] Jensen, Christian S. ; Dyreson, Curtis E. ; Bohlen, Michael ; Clifford, James ; Elmasri, Ramez ; Gadia, Shashi K. ; Grandi, Fabio ; Hayes, Pat ; Jajodia, Sushil ; Käfer, Wolfgang ; Kline, Nick ; Lorentzos, Nikos ; Mitsopoulos, Yannis ; Montanari, Angelo ; Nonen, Daniel ; Peressi, Elisa ; Pernici, Barbara ; Roddick, John F. ; Sarda, Nandlal L. ; Scalas, Maria Rita ; Segev, Arie ; Snodgrass, Richard T. ; Soo, Mike D. ; Tansel, Abdullah ; Tiberio, Paolo ; Wiederhold, Gio: The consensus glossary of temporal database concepts - february 1998 version. In: *Lecture Notes in Computer Science* 1399 (1998), S. 367–405. – ISSN 0302–9743

[JG08] Jacobi, Jonas ; Grawunder, Marco: ODYSSEUS: Ein flexibles Framework zum Erstellen anwendungsspezifischer Datenstrommanagementsysteme. In: *Grundlagen von Datenbanken* 1 (2008), S. 86–90

[JG10] Junghans, Conny ; Gertz, Michael: Modeling and Prediction of Moving Region Trajectories. In: *Proceedings of the ACM SIGSPATIAL International Workshop on GeoStreaming*. New York, NY, USA : ACM, 2010 (IWGS '10). – ISBN 978–1–4503–0431–3, 23–30

[Kar13] Karney, Charles F. F.: Algorithms for geodesics. In: *Journal of Geodesy* 87 (2013), Jan, Nr. 1, 43–55. http://dx.doi.org/10.1007/s00190-012-0578-z. – DOI 10.1007/s00190–012–0578–z. – ISSN 1432–1394

[Krä07] Krämer, Jürgen ; Seeger, Bernhard (Prof. D. (Hrsg.): *Continuous Queries over Data Streams - Semantics and Implementation*. Philipps-Universität Marburg, 2007 http://archiv.ub.uni-marburg.de/diss/z2007/0671/pdf/djk.pdf

[KS04a] Kolahdouzan, Mohammad R. ; Shahabi, Cyrus: Continuous K-Nearest Neighbor Queries in Spatial Network Databases. In: *GeoInformatica* 9 (2004), S. 321–341

[KS04b] Krämer, Jürgen ; Seeger, Bernhard: PIPES: A Public Infrastructure for Processing and Exploring Streams. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2004 (SIGMOD '04). – ISBN 1–58113–859–8, 925–926

[KS09] Krämer, Jürgen ; Seeger, Bernhard: Semantics and Implementation of Continuous Sliding Window Queries over Data Streams. In: *ACM Trans. Database Syst.* 34 (2009), April, Nr. 1, 4:1–4:49. http://dx.doi.org/10.1145/1508857.1508861. – DOI 10.1145/1508857.1508861. – ISSN 0362–5915

[KSF+03] Koubarakis, Manolis ; Sellis, Timos ; Frank, Andrew U. ; Grumbach, Stéphane ; Güting, Ralf H. ; Jensen, Christian S. ; Lorentzos, Nikos ; Manolopoulos, Yannis ; Nardelli, Enrico ; Pernici, Barbara u. a.: *Spatio-temporal databases: The CHOROCHRONOS approach*. Bd. 2520. Springer, 2003

[Kuk15] Kuka, Christian: *Qualitätssensitive Datenstromverarbeitung zur Erstellung von dynamischen Kontextmodellen*, Universität Oldenburg, Diss., 2015

[KWZH14] Kuwata, Y. ; Wolf, M. T. ; Zarzhitsky, D. ; Huntsberger, T. L.: Safe Maritime Autonomous Navigation With COLREGS, Using Velocity Obstacles. In: *IEEE Journal of Oceanic Engineering* 39 (2014), Jan, Nr. 1,

S. 110–119. http://dx.doi.org/10.1109/JOE.2013.2254214. – DOI 10.1109/JOE.2013.2254214. – ISSN 0364–9059

[LCY07] Lin, Dan ; Cui, Bin ; Yang, Dongqing: Optimizing Moving Queries over Moving Object Data Streams. In: Kotagiri, Ramamohanarao (Hrsg.) ; Krishna, P. R. (Hrsg.) ; Mohania, Mukesh (Hrsg.) ; Nantajeewarawat, Ekawit (Hrsg.): *Advances in Databases: Concepts, Systems and Applications: 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007. Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007. – ISBN 978–3–540–71703–4, 563–575

[LGMR10] Longley, Paul A. ; Goodchild, Mike ; Maguire, David J. ; Rhind, David W.: *Geographic Information Systems and Science*. 3rd. Wiley Publishing, 2010. – ISBN 0470721448, 9780470721445

[LH06] Lin, Bin ; Huang, Chih-Hao: Comparison between ARPA Radar and AIS Characteristics for Vessel Traffic Services. In: *Journal of Marine Science and Technology* 14 (2006), Sep, Nr. 3, S. 182–189. – ISSN 1023–2796

[Loc] LocationTech: *locationtech jts*. https://github.com/locationtech/jts

[LTS+08] Li, Jin ; Tufte, Kristin ; Shkapenyuk, Vladislav ; Papadimos, Vassilis ; Johnson, Theodore ; Maier, David: Out-of-order processing: a new architecture for high-performance stream systems. In: *PVLDB* 1 (2008), Nr. 1, 274–288. http://dx.doi.org/10.14778/1453856.1453890. – DOI 10.14778/1453856.1453890

[Luc02] Luckham, David C.: *The Power of Events - An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Amsterdam : Addison-Wesley, 2002. – ISBN 978–0–201–72789–0

[Lud15] Ludmann, Cornelius A.: Online Recommender Systems Based on Data Stream Management Systems. In: *Proceedings of the 9th ACM Conference on Recommender Systems*. New York, NY, USA : ACM, 2015 (RecSys '15). – ISBN 978–1–4503–3692–5, 391–394

[Lud17] Ludmann, Cornelius A.: Recommending News Articles in the CLEF News Recommendation Evaluation Lab with the Data Stream Management System Odysseus. In: Cappellato, Linda (Hrsg.) ; Ferro, Nicola (Hrsg.) ; Goeuriot, Lorraine (Hrsg.) ; Mandl, Thomas (Hrsg.): *Working Notes of CLEF 2017 - Conference and Labs of the Evaluation Forum, Dublin, Ireland, September 11-14, 2017*. Bd. 1866, CEUR-WS.org, 2017 (CEUR Workshop Proceedings)

[MA08] Mokbel, Mohamed F. ; Aref, Walid G.: SOLE: scalable on-line execution of continuous queries on spatio-temporal data streams. In: *The VLDB Journal* 17 (2008), Nr. 5, S. 971–995

[MFBM14] McKenney, Mark ; Frye, Roger ; Benchly, Zachary ; Maughan, Logan: Temporal coverage aggregates over moving region streams. In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on GeoStreaming* ACM, 2014, S. 21–24

[OGC] OGC: *Welcome to the OGC*. http://www.opengeospatial.org/

[Oos99] Oosterom, Peter van: Spatial Access Methods. In: *Encyclopedia of Database Systems*, 1999

[PAA+17] Patroumpas, Kostas ; Alevizos, Elias ; Artikis, Alexander ; Vodas, Marios ; Pelekis, Nikos ; Theodoridis, Yannis: Online event recognition from moving vessel trajectories. In: *GeoInformatica* 21 (2017), Nr. 2, 389–427. http://dx.doi.org/10.1007/s10707-016-0266-x. – DOI 10.1007/s10707–016–0266–x. – ISSN 1573–7624

[PTRC07] Peffers, Ken ; Tuunanen, Tuure ; Rothenberger, Marcus A. ; Chatterjee, Samir: A design science research methodology for information systems research. In: *Journal of management information systems* 24 (2007), Nr. 3, S. 45–77

[PVB13] Pallotta, Giuliana ; Vespe, Michele ; Bryan, Karna: Vessel pattern knowledge discovery from ais data: A framework for anomaly detection and route prediction. In: *Entropy* 15 (2013), Nr. 6, S. 2218–2245

[RH10] Roh, Gook-Pil ; Hwang, Seung-won: NNCluster: An Efficient Clustering Algorithm for Road Network Trajectories. In: *Proceedings of the 15th International Conference on Database Systems for Advanced Applications - Volume Part II*. Berlin, Heidelberg : Springer-Verlag, 2010 (DASFAA'10). – ISBN 3–642–12097–0, 978–3–642–12097–8, 47–61

[RSV02] Rigaux, Ph ; Scholl, Michel O. ; Voisard, Agnès: *Spatial Databases - With Application to GIS*. San Francisco, Calif : Morgan Kaufmann, 2002. – ISBN 978–1–558–60588–6

[Sal14] Salonen, Joni: *Geographic distance can be simple and fast*. http://jonisalonen.com/2014/computing-distance-between-coordinates-can-be-simple-and-fast. Version: 2014

[SM07] Schwehr, K. D. ; McGillivary, P. A.: Marine Ship Automatic Identification System (AIS) for Enhanced Coastal Security Capabilities: An Oil Spill Tracking Application. In: *OCEANS 2007*, 2007. – ISSN 0197–7385, S. 1–9

[SRC+16] Schmidt, Desmond ; Radke, Kenneth ; Camtepe, Seyit ; Foo, Ernest ; Ren, Michał: A survey and analysis of the GNSS spoofing threat and countermeasures. In: *ACM Computing Surveys (CSUR)* 48 (2016), Nr. 4, S. 64

[SSBK12] Schmiegelt, Philip ; Seeger, Bernhard ; Behrend, Andreas ; Koch, Wolfgang: Continuous Queries on Trajectories of Moving Objects. In: *Proceedings of the 16th International Database Engineering & Applications Sysmposium*. New York, NY, USA : ACM, 2012 (IDEAS '12). – ISBN 978–1–4503–1234–9, 165–174

[Sto03] Stolze, Knut: SQL/MM Spatial-The Standard to Manage Spatial Data in a Relational Database System. In: *BTW* Bd. 2003, 2003, S. 247–264

[SV04] Schiller, Jochen ; Voisard, Agnès: *Location-Based Services*. Amsterdam : Elsevier, 2004. – ISBN 978–0–080–49172–1

[SW04] Srivastava, Utkarsh ; Widom, Jennifer: Flexible Time Management in Data Stream Systems. In: *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, NY, USA : ACM, 2004 (PODS '04). – ISBN 158113858X, 263–274

[THC+14] Tseng, P. J. ; Hung, C. C. ; Chuang, Y. H. ; Kao, K. ; Chen, W. H. ; Chiang, C. Y.: Scaling the Real-Time Traffic Sensing with GPS Equipped Probe Vehicles. In: *2014 IEEE 79th Vehicular Technology Conference (VTC Spring)*, 2014. – ISSN 1550–2252, S. 1–5

[TMRDR12] Tao, Sha ; Manolopoulos, Vasileios ; Rodriguez Duenas, Saul ; Rusu, Ana: Real-Time Urban Traffic State Estimation with A-GPS Mobile Phones as Probes. In: *Journal of Transportation Technologies* 2 (2012), Nr. 1, 22–31. http://dx.doi.org/10.4236/jtts.2012.21003. – DOI 10.4236/jtts.2012.21003. – QC 20120312

[Tso16] Tsou, Ming-Cheng: Online analysis process on Automatic Identification System data warehouse for application in vessel traffic service. In: *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment* 230 (2016), Nr. 1, 199-215. http://dx.doi.org/10.1177/1475090214541426. – DOI 10.1177/1475090214541426

[Vin75] Vincenty, Thaddeus: Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. In: *Survey review* 23 (1975), Nr. 176, S. 88–93

[VVBB12] Vespe, M. ; Visentini, I. ; Bryan, K. ; Braca, P.: Unsupervised learning of maritime traffic patterns for anomaly detection. In: *9th IET Data Fusion Target Tracking Conference (DF TT 2012): Algorithms Applications*, 2012, S. 1–5

[WK96] Widmer, Gerhard ; Kubat, Miroslav: Learning in the Presence of Concept Drift and Hidden Contexts. In: *Machine Learning* 23 (1996), Apr, Nr. 1, 69–101. http://dx.doi.org/10.1023/A:1018046501280. – DOI 10.1023/A:1018046501280. – ISSN 1573–0565

[WKK+14] Wakamiya, Shoko ; Kawai, Yukiko ; Kawasaki, Hiroshi ; Lee, Ryong ; Sumiya, Kazutoshi ; Akiyama, Toyokazu: Crowd-sourced prediction of pedestrian congestion for bike navigation systems. In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on GeoStreaming* ACM, 2014, S. 25–32

[WLN14] Whittier, J. C. ; Liang, Qinghan ; Nittel, Silvia: Evaluating Stream Predicates over Dynamic Fields. In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on GeoStreaming*. New York, NY, USA : ACM, 2014 (IWGS '14). – ISBN 978–1–4503–3139–5, 2–11

[WNPL13] Whittier, J. C. ; Nittel, Silvia ; Plummer, Mark A. ; Liang, Qinghan: Towards Window Stream Queries over Continuous Phenomena. In: *Proceedings of the 4th ACM SIGSPATIAL International Workshop on GeoStreaming*. New York, NY, USA : ACM, 2013 (IWGS '13). – ISBN 978–1–4503–2532–5, 2–11

[YWS15] Yu, Jia ; Wu, Jinxuan ; Sarwat, Mohamed: 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*. New York, NY, USA : ACM, 2015 (SIGSPATIAL '15). – ISBN 978–1–4503–3967–4, 70:1–70:4

[ZJDR10] Zhang, Rui ; Jagadish, H.V. ; Dai, Bing T. ; Ramamohanarao, Kotagiri: Optimized algorithms for predictive range and KNN queries on moving objects. In: *Information Systems* 35 (2010), Nr. 8, 911 - 932. http://dx.doi.org/https://doi.org/10.1016/j.is.2010.05.004. – DOI https://doi.org/10.1016/j.is.2010.05.004. – ISSN 0306–4379

[ZZP+03] Zhang, Jun ; Zhu, Manli ; Papadias, Dimitris ; Tao, Yufei ; Lee, Dik L.: Location-based Spatial Queries. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2003 (SIGMOD '03). – ISBN 1–58113–634–X, 443–454

Index

Symbols

A

B

C

D

E

F

G

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis an der Carl von Ossietzky Universität Oldenburg und den DFG-Richtlinien festgelegt sind, befolgt habe. Des Weiteren habe ich im Zusammenhang mit dem Promotionsvorhaben keine kommerziellen Vermittlungs- oder Beratungsdienste in Anspruch genommen.

Oldenburg, den 3. März 2020

Tobias Brandt