UNIVERSITÉ
LIBRE
DE BRUXELLES

JMEOS: A Java binding of the MEOS spatiotemporal library

Master Thesis

Mareghni Nidhal

Master thesis submitted under the supervision of
Prof. Esteban Zimányi

Academic year
2023-2024

In order to be awarded the Master's programme in
Computer Science

Abstract

The increasing complexity and volume of spatiotemporal data in various domains necessitate efficient and accessible tools for data handling and analysis. MobilityDB, an open-source moving object database, has established itself as a pioneer tool in this landscape. However, with the emergence of big data, there's an urgent requirement to exploit MobilityDB's capabilities, through widely used programming languages, such as Python and Java. It's within this context that JMEOS, a Java-based library, becomes relevant. It bridges the gap between Java applications and MobilityDB, allowing for the seamless integration of advanced temporal types and functionalities. The main focus of this thesis is the implementation of a Java binding of the MEOS library. To this end, we map the functionalities through JNR-FFI, a popular C foreign function interface. By means of it, we implement analogous MobilityDB spatiotemporal types such as `TBox`, `FloatSpan`, `PeriodSet` or `TGeomPoint`. Lastly, we perform unit tests and code analysis to ensure the functionality and reliability of JMEOS . We finish the thesis by implementing use case example to demonstrate its efficacy in real-world scenarios and benchmarking its performance against MobilityDB and MEOS.

Keywords— JAVA, PostgreSQL, MobilityDB, Spatiotemporal, MEOS

Acknowledgements

Firstly, I would like to acknowledge all my gratitude to the Professor Esteban Zimányi for his constant availability during meetings, his important advices from the beginning to the end of the project and also for his precise and kind supports on the status of MobilityDB and PyMEOS.
I would like to also thank Killian Monnier who helped me brainstorm and implement some important part of this project.
Lastly, I want to thank my friends and family who supported me and provided me multiple points of views as well as feedbacks during the implementation of the project and the redaction of the thesis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

In the current context of global warming and the need to reduce greenhouse gas emissions, public transport is one of the most sustainable solutions to environmental problems. To take full advantage of this and avoid the opposite effect, it is vital to optimize public transport systems in terms of cost, routes and distribution strategies. distribution strategies. Optimizing these networks, which have become increasingly complex and extensive, encompassing thousands of elements such as rolling stock or support and logistics vehicles, requires precise and rapid data management and analysis.

In this context, traditional databases are often limited, mainly due to their inability to efficiently process complex spatiotemporal data, i.e. speed of a vehicle at a certain moment in time within a specific position. These data, which combine the spatial (where things happen) and temporal (when they happen) aspects, are essential for precise management and analysis of movements and trajectories in transport systems, object or vehicle monitoring, etc.

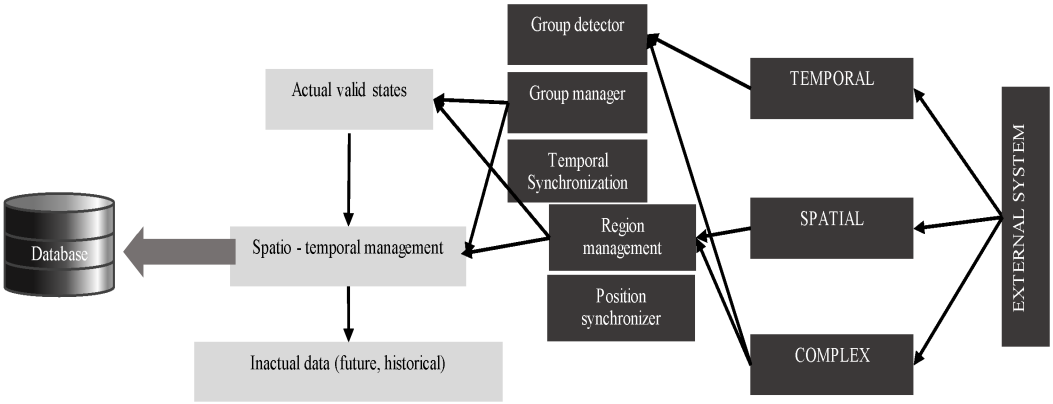

Figure 1.1: Example of Temporal and Spatial Data Managed Separately on a Classical Database [21].

This is where MobilityDB comes in as a mainstream Moving Object Database (MOD) solution. It gathers, among others, more than 2300 functions, more than 40 extended types and 67,000 lines of C code. Built on the solid foundations of PostgreSQL and PostGIS, this open source system is focused on the management and analysis of geospatial data, in particular that relating to moving objects and trajectories. MobilityDB is effi-

cient and expressive MOD based on operations, indexing, aggregation, and optimization framework, offering a powerful tool for Moving Objects.

However, the accessibility and use of MobilityDB can be limited by the need to master specific technologies such as SQL or C, the main programming language used for its core library called MEOS (Mobility Engine, Open Source). To overcome this barrier and broaden access to types and functions, PyMEOS the Python implementation of MEOS library was first developed and recently released.

Analogously, JMEOS represents a Java implementation of the MEOS library and aims to make the manipulation of temporal and spatiotemporal data more accessible to developers and analysts who are more familiar with the Java environment. By bringing the capabilities of MEOS to the Java world, JMEOS opens the door to wider integration and more flexible use of MobilityDB in a variety of applications.

It is in this context, of the spread of the MobilityDB system, that this thesis presenting JMEOS is set.

1.1.1 PostgreSQL

Since MobilityDB is an extension of PostgreSQL, it is necessary to present a brief background knowledge on this system. PostgreSQL, an advanced open-source object-relational database system, stands out for its robustness, scalability, and compliance with SQL standards. Originating from the POSTGRES project at the University of California, Berkeley, it has evolved significantly over the years, supporting complex data types and sophisticated programming interfaces [41].

One of its main advantages is the support for SQL and JSON querying, which makes it versatile for various data types [28]. It is highly extensible, allowing users to define their own data types, index types or functional languages. PostgreSQL's powerful mechanism for handling concurrency, Multi-Version Concurrency Control (MVCC), ensures data integrity and performance in high-volume environments [19].

PostgreSQL supports various types of indexes, like B-tree, Hash, GiST, SP-GiST, and GIN, each optimized for different kinds of queries and data patterns [31]. For example, B-tree indexes are well-suited for high-cardinality data and can greatly accelerate equality and range queries. GiST and SP-GiST indexes are particularly useful for full-text search and geospatial data, which are essential in MobilityDB

PostgreSQL also ensures robust data integrity through features like complex constraint checking, foreign keys, and transactional integrity, thereby making it suitable for applications that require reliable and secure data management [23] compared to other relational database system such as MySQL, as shown on Table 1.1. Furthermore, its active open-source community contributes to its continuous improvement, providing regular updates and extensive documentation [36].

PostgreSQL Advantages	**MySQL Comparison**
Advanced SQL Compliance	Less compliant with SQL standards
Richer Set of Features	More basic feature set
Extensive Indexing Options	Limited indexing options
Better Support for Complex Queries	Less suited for complex queries
Robust and Extensible	Less extensibility
Stronger Emphasis on Data Integrity	Weaker data integrity by default
More Advanced Geospatial Support (PostGIS)	Basic geospatial features
Support for Custom Data Types	Limited custom data types support
Higher Concurrency without Locking Issues	Locking issues at high concurrency
More Suitable for Large and Complex Databases	Better for simpler, read-heavy workloads

Table 1.1: Comparative Advantages of PostgreSQL over MySQL

1.1.2 PostGIS

PostGIS, an extension of the PostgreSQL relational database, is designed to manage and analyze geographical data efficiently, making it a powerful tool for Geographic Information Systems (GIS). This extension enhances PostgreSQL by enabling it to store and process geospatial data, adhering to the Open Geospatial Consortium (OGC) standards [26].

One of the major advantages of PostGIS is its spatial indexing capability, which significantly speeds up geospatial queries. This feature is particularly beneficial for applications dealing with large datasets, such as urban planning and environmental management [4]. PostGIS supports a wide array of spatial functions, including advanced querying for geospatial analytics, and is compatible with numerous GIS software, facilitating seamless data exchange and interoperability [47].

PostGIS supports a diverse range of geospatial data types and operations, enhancing PostgreSQL's capabilities in spatial data handling. Key data types include geometry, geography, raster, and topology, which allow for the representation of `points`, `lines`, `polygons`, and `raster` images in a spatial context [26]. It also supports spatial operations like spatial joins, nearest neighbor searches, and network routing, enabling complex geospatial analyses [4]. PostGIS can handle spatial predicates (such as intersects, contains, and touches) and spatial functions for area, distance, and perimeter calculations [29]. Additionally, it offers advanced functionalities sich as 3D and 4D indexing, which are crucial for temporal-spatial data applications [29].

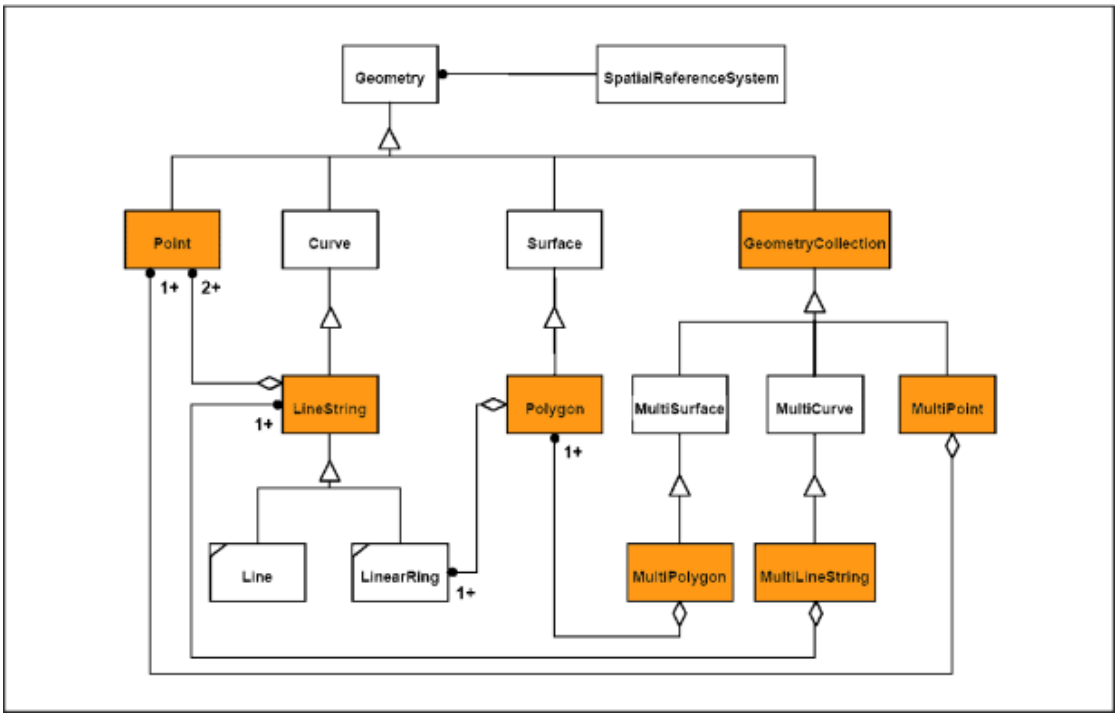

Figure 1.2: PostGIS Geometry Hierarchy [37].

1.1.3 MobilityDB

MobilityDB, a new mainstream Moving Object Database System, is a database extension for PostgreSQL and PostGIS, specifically designed to manage and analyze spatiotemporal data, primarily focusing on movement or trajectory data. It extends the capabilities of PostgreSQL and PostGIS to handle time-dependent geospatial data effectively.

MobilityDB adds several crucial data types, such as temporal points (tgeompoint, tgeogpoint) or temporal sequence numbers (tintseq, tfloatseq), shown in Figure 1.3 which are essential for representing moving objects over time. These data types are fundamentally built on bool, int, float, and text types of PostgreSQL and the spatial geometry/geography types of PostGIS, therefore combining the temporal aspect with spatial information [46].

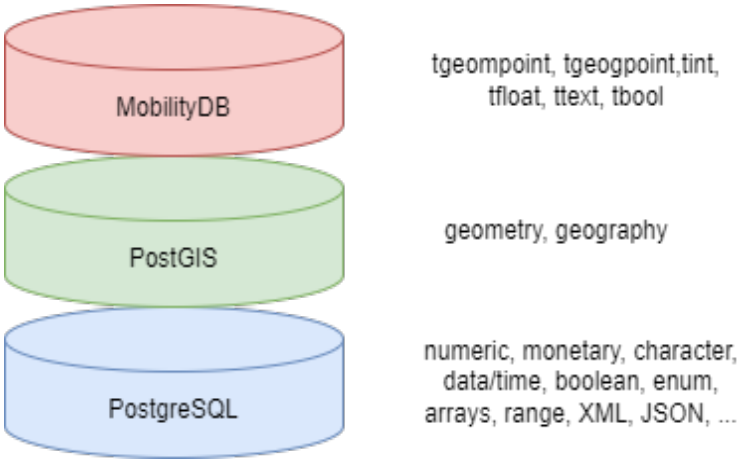

Figure 1.3: MobilityDB Relation with PostGIS and PostgreSQL [46].

MobilityDB also supports a rich set of temporal predicates and functions, such as temporal filtering, aggregation, and joins, which are crucial for analyzing spatiotemporal datasets [1], as shown in Figure 1.4 . Temporal functions and operators in MobilityDB focus on the temporal aspect of values, which could be instants, ranges, arrays of instants, or arrays of ranges. The atPeriods function limits a temporal type to specified time ranges, while duration extracts a value's defined time.

Lifted functions and operators in MobilityDB intuitively extend operators from base types (like arithmetic operations for integers and floats, spatial relations, and distances for geometries) to accommodate evolving time values. MobilityDB's spatiotemporal functions expand on PostGIS's spatial functions for both `geometry` and `geography` types, handling temporal elements and relying on PostGIS for spatial computations. Spatiotemporal functions and operators comprise the remaining category. Examples include calculating speed and azimuth for `tgeompoint`/`tgeogpoint`, maxValue for `tfloat`/`tint`, and twAvg for `tfloat` [46].

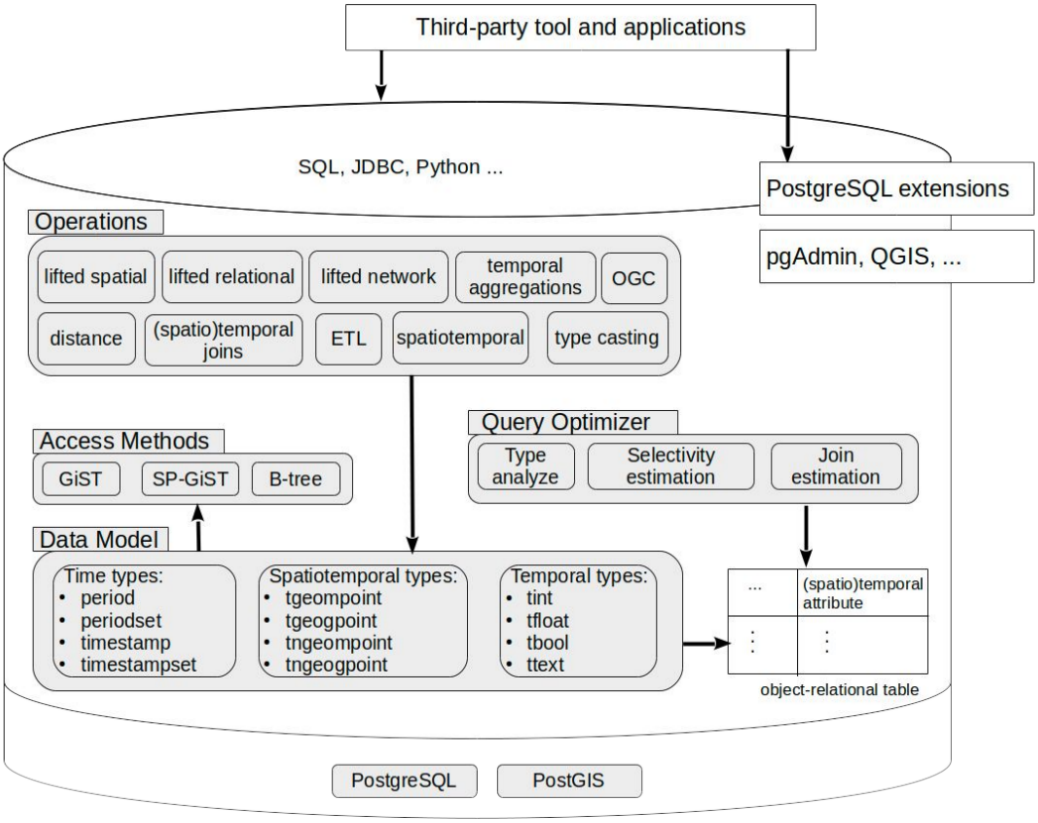

Figure 1.4: MobilityDB Overall Architecture [46].

A key feature of MobilityDB is its efficient handling of large spatiotemporal datasets. The GiST index employs an R-tree for managing temporal alphanumeric types and utilizes a TB-tree for temporal point types. Conversely, the SP-GiST index uses a Quad-tree for handling temporal alphanumeric types and an Oct-tree for temporal point types. Thus, MobilityDB's functionalities are specifically optimized for typical mobility data queries, such as retrieving the trajectory of an object within a specific time frame or computing the distance traveled by a moving object.

The extension is particularly advantageous for transportation planning and traffic management, where understanding the movement patterns of vehicles and people is crucial. It allows for advanced analyses like congestion detection, travel time estimation, and route optimization, which are essential for smart city initiatives and sustainable urban planning [35].

1.1.4 MobilityDB-python and MobilityDB-JDBC

MobilityDB-python [5] was a database adapter designed to facilitate access to MobilityDB functionalities from the Python programming language.

The structure of MobilityDB-python revolved around providing Pythonic functions and classes that correspond to the data types and operations available in MobilityDB. Among these data types and operations, we can cite `TInstant`, `TSequence` or `TSequenceSet` as shown in the Figure 1.5:

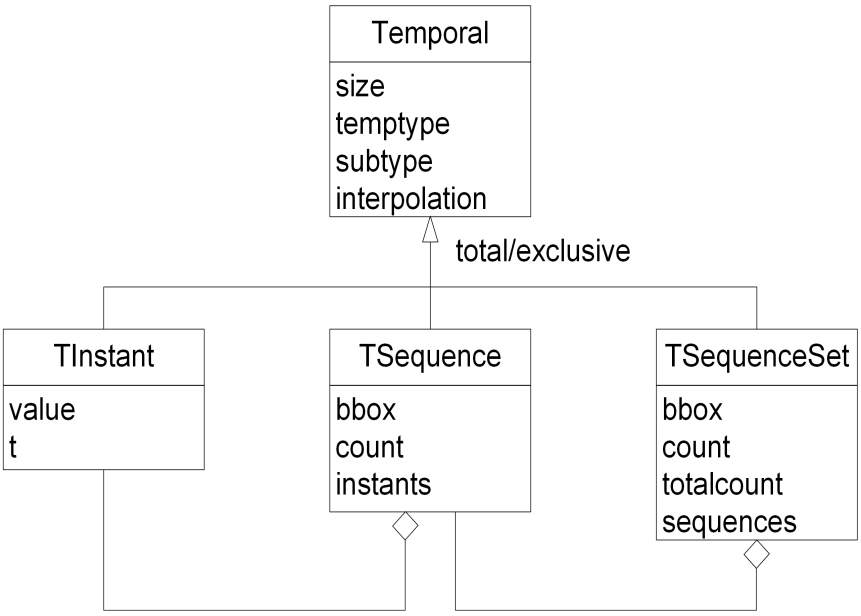

Figure 1.5: MobilityDB-python Temporal Architecture [13].

This structure allowed Python developers to work with complex spatiotemporal queries and data types, such as temporal points and periods, using familiar syntax and programming paradigms.

This adapter used popular PostgreSQL connectors like psycopg2 for synchronous operations and asyncpg for asynchronous support, ensuring compatibility with different Python programming requirements. Furthermore, it leveraged the postgis adapter to interact with PostGIS functionalities, essential for handling geometric and geographic data. Currently, MobilityDB-ptyhon is deprecated and evolved into PyMEOS.

Similarly, the Java Database Connectivity Driver for MobilityDB [30] enhances accessibility and simplifies integration from Java, one of the most globally utilized programming languages. This development aims to match the capabilities provided by the pre-existing MobilityDB-Python adapter.

The main disadvantage of both adapters resides in the fact that a database is previously needed in order to utilize the implemented functions and types.

1.1.5 PyMEOS and JMEOS

PyMEOS library, standing for Python MEOS, serve as the legitimate replacement of MobilityDB-Python database adapter which is currently deprecated. The motivation

behind this project is to enable the utilization of MobilityDB, through the MEOS library, without the needs of a preset database and in a streaming fashion, as shown in Figure 1.6 . This corresponds more to a larger public and its current trends of data manipulation, i.e. through CSV files.

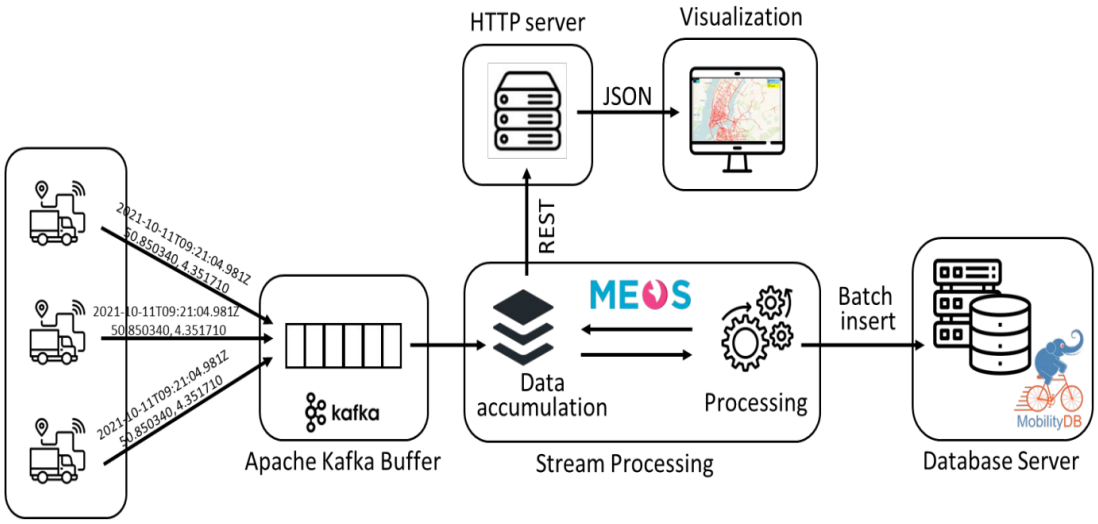

Figure 1.6: Mobility Data in Stream and Edge Processing [43].

PyMEOS inner layer is based on CFFI, the C Foreign Function Interface for Python, which acts as a bridge between Python and MEOS. The upper layer wraps the CFFI and declares a set of operations that extracts and declares corresponding utility functions obtained from MEOS inner code. The last layer of PyMEOS comprises the core of the library in which all types and their underlying operations (using obtained utility functions) are implemented. Figure 1.7 displays the structure of PyMEOS.

Analogously, JMEOS will intend as the replacement of the MobilityDB-JDBC adapter and will tend to be a JAVA equivalence of PyMEOS.

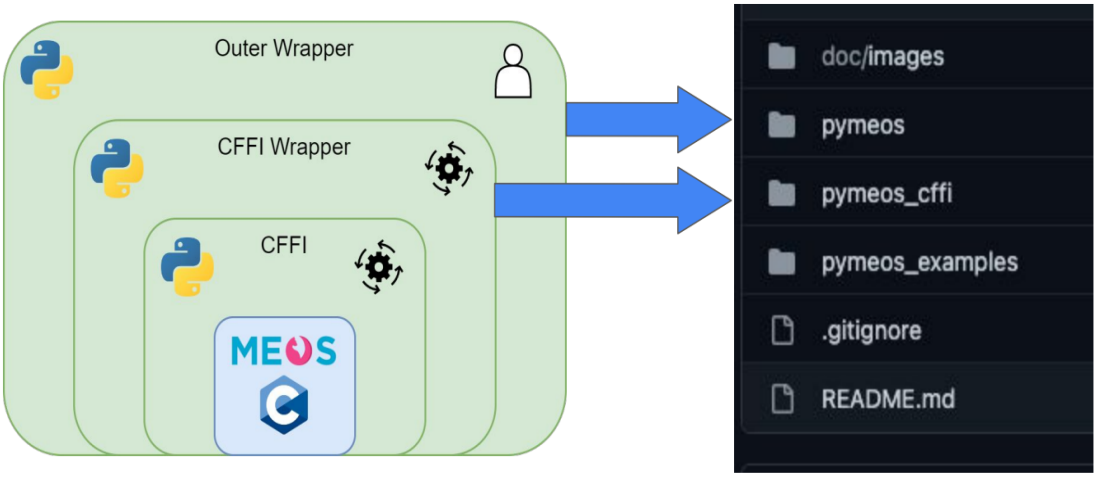

Figure 1.7: PyMEOS Wrapper Architecture [44].

1.2 Structure and Objective of the Thesis

As previously mentioned, JMEOS wishes to be an equivalent implementation of PyMEOS with its own CFFI wrapper and its own spatiotemporal data types realization in alignment with MobilityDB types. The thesis is structured in the following manner:

- Chapter 2 describes the requirements and the systems used during the implementation, the testing and the deployment of JMEOS.
- Chapter 3 covers the design of the project. It also presents the extraction function implementation from the MEOS source code and the spatiotemporal types (inherited from MobilityDB) implementation. This chapter also concerns the deployment, through Docker, of the project.
- Chapter 4 presents the unit tests performed on a substantial majority of the code.
- Chapter 5 discusses about the quality of the code analyzed thanks to the application SonarQube.
- Chapter 6 provides a project's execution of some MEOS examples that manipulates real-case data on which benchmarks will be performed with comparisons between JMEOS, MEOS and PyMEOS
- Chapter 7 provides an overall conclusion of the project presented in this thesis.

Concerning the objectives, we can pinpoint the main ones:

1. Create a Java binding of the MEOS library
2. Design Java data structures correlated with MobilityDB spatiotemporal types.
3. Implement a set of spatiotemporal data types and corresponding functions matching MEOS implementation.
4. Test the implementation and the project through unit test, example files with AIS [39] data.
5. Assess the code quality with a tool and propose a deployment of the project on an isolated environment.

1.3 Terms and Notations

Throughout this thesis, multiple terms and notations in relation with MobilityDB, are employed. This subsection aims to give a clear definition of these notions in order to improve the reading comprehension.

- The MobilityDB (and JMEOS) notation for temporal types is denoted with the letter `t` followed by the base type `tbox, tfloat, tbool, ..`, the duration type `tinstant, tsequence, tsequenceset` or a mix of both `tfloatinst, ttextseq, tintseqset, ...`
- Types and functions related to MobilityDB, MEOS, PyMEOS, JMEOS will be each depicted in a specific font.
- In the same manner, a moving point is expressed as a geometric point with `tgeompoint` or a geographic point with `tgeogpoint`.
- Finally, MEOS [13] standing for **Mobility Engine, Open Source**, the C library being a core component of MobilityDB. JDBC [30] expresses **Java Database Connectivity Driver for MobilityDB**, a tool for connecting Java applications to various MobilityDB databases. **BerlinMOD** [12], a benchmark suite for evaluating MobilityDB spatiotemporal databases.

Chapter 2

Requirements and System Used

The development of JMEOS was conducted on Linux Ubuntu 22.04 LTS due to the current stable version of MobilityDB being operational exclusively on Linux. The Windows version of MobilityDB is still under development, and the exclusive way to access it on Windows platforms is through Docker. For this project, openjdk 21 was utilized, ensuring a robust and reliable development environment. Additionally, IntelliJ IDEA was the chosen Integrated Development Environment (IDE) for its numerous advantages:

- **Advanced Code Navigation:** IntelliJ provides efficient and intuitive code navigation features, enhancing productivity.
- **Intelligent Code Completion:** The IDE offers context-aware code completion, speeding up the coding process.
- **Built-in Tools and Integration:** IntelliJ supports a wide range of built-in development tools and seamless integration with other software.
- **User-Friendly Interface:** The IDE's interface is designed for ease of use, improving the overall development experience.

Dependency	**Version**
Ubuntu	22.04 LTS
openjdk	21
IntelliJ IDEA	2023.3.1

Table 2.1: Development Dependencies and Their Versions for JMEOS

As one of our objectives is to make the library accessible to JAVA users, we decided to use Maven, an automatic project production manager. Maven makes it easy to interweave the various dependencies, tests and construction/deployment phases of the library.

In that manner, this project relies on a few external dependencies to facilitate the implementation of the various classes and functions presented in Chapter 3 . In order to submit the library more deployable, less prone to potential errors and faster to use (primary objective) , it was decided to reduce the number of these dependencies as much as possible, and to keep only those that are essential to the operation of JMEOS. These include JNR-FFI, JTS-Core, Guava and Junit.

Ultimately, Docker will enable you to create a project image to avoid any compatibility problems between JMEOS dependencies and the machine configurations on which

the library will be operated. In the same way as MobilityDB, Docker will allow Windows users to access the library's functionality.

2.1 MobilityDB and MEOS

As the basis of the project is the MEOS library, it goes without saying the first requirement of the project is the MEOS library. As this is the heart of MobilityDB, it must be installed on the machine from which you wish to develop or run the JMEOS library internally. Details of installation instructions are available on the official MobilityDB project website: https://mobilitydb.com/install.html

The various requirements for this installation are listed in the Table 2.2

Requirements	**Version**
PostgreSQL	>= 11
CMake	>= 3.7
PostGIS	>= 2.5
JSON-C	
GNU Scientific Library (GSL)	
Development files for PostgreSQL, PostGIS/liblwgeom, PROJ and JSON-C	

Table 2.2: System Requirements and Versions

```
1 john@doe:~$ apt install build-essential cmake
    ↪ postgresql-server-dev-11 \
2 liblwgeom-dev libproj-dev libjson-c-dev libprotobuf-c-dev
3 john@doe:~$ git clone
    ↪ https://github.com/MobilityDB/MobilityDB
4 john@doe:~$ mkdir MobilityDB/build
5 john@doe:~$ cd MobilityDB/build
6 john@doe:~$ cmake -DMEOS=ON ..
7 john@doe:~$ make
8 john@doe:~$ sudo make install
```

Listing 2.1: Linux MobilityDB Installation Code

The setup for MobilityDB on a Linux system begins by installing necessary packages using the depicted command, which ensures all essential build tools and libraries are in place. Next, the MobilityDB repository is cloned from GitHub to the local machine.

A separate build directory is created within the cloned repository to maintain compiled files organized. The build environment, including MEOS support, is then configured with the corresponding flag Following this, make compiles the MobilityDB code, and sudo make install finalizes the installation, making MobilityDB available for use on

the system.

Once this step has been completed, the MEOS library will be present in the **build** installation folder, and the **libmeos.so** file representing the library compiled in the shared libraries folder of the machine currently running.

The Figure 2.1 describes the MobilityDB repository with the interactions between MobilityDB, MEOS and PostGIS where each directory contains specific code but where the MEOS library borrows code from PostGIS, and MobilityDB borrows code from all.

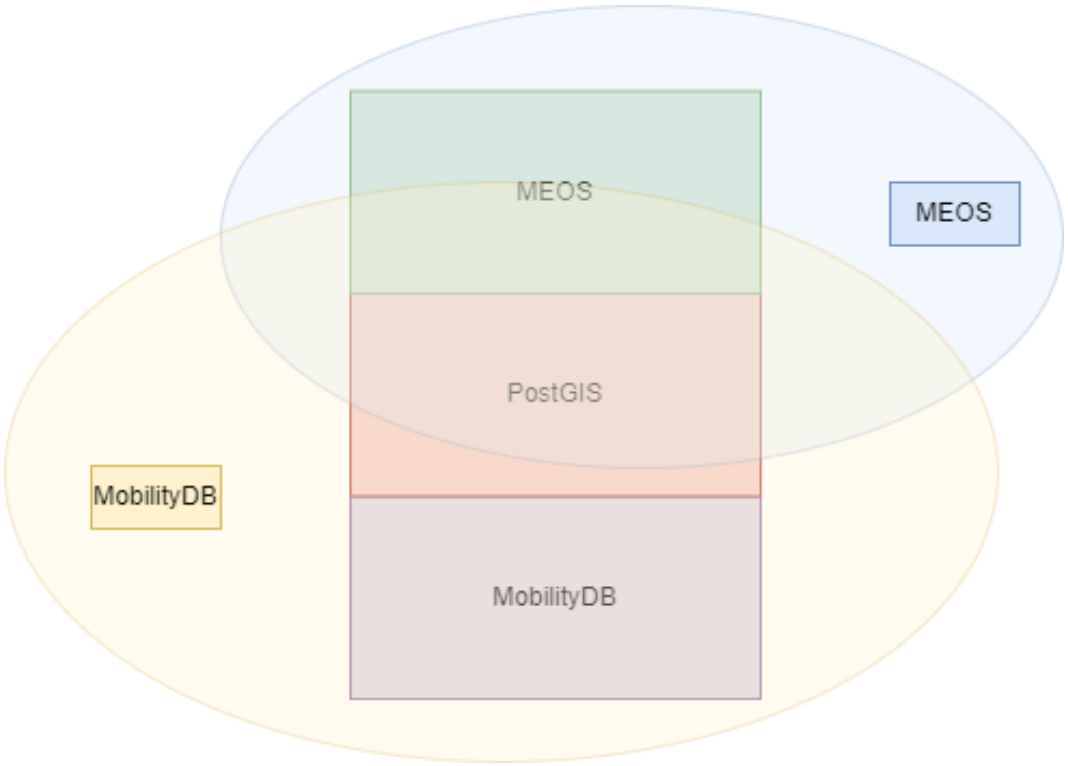

Figure 2.1: Codebase of MEOS and MobilityDB [45].

2.2 Maven and Dependencies

This section displays the few dependencies and their version, that are integrated through Maven, during the development of JMEOS.

Requirements	Version
jnr-ffi	2.2.14
maven-plugin-plugin	3.10.2
junit-jupiter (test scope)	5.10.0
guava	30.0-android
jts-core	1.18.2

Table 2.3: Project Dependencies and Their Versions

- JNR-FFI is the C foreign function interface that is necessary to interact with MEOS C library.
- Maven plugins will be extremely useful to build the project in a jar file, to generate the javadocs and to add plugins for specific action (tests).
- JUnit is the framework responsible to offer and execute the unit test interface in JMEOS.
- Guava provides ranges functionalities that are not available in JAVA. The corresponding library in PyMEOS is spans.
- JTS-Core covers geometry features such as polygon, line and point that will be necessary in the development of some JMEOS features. The corresponding library in PyMEOS is **Shapely**.

2.3 C Foreign Function Interface

A C Foreign Function Interface (CFFI) represent a mechanism that allows a higher-level programming language like Java to call functions and use data structures defined in a lower-level language like C. This interface is crucial in software development when there's a need to integrate or utilize libraries written in different programming languages.

In the context of JMEOS, which is a Java implementation of the MEOS library, CFFI is essential for several reasons:

- **Interoperability Between Java and C:** MEOS is written in C, and JMEOS is implemented in Java. Since Java and C operate in fundamentally different runtime environments, CFFI serves as a bridge, enabling JMEOS to interact with the MEOS library. This interaction is necessary to leverage MEOS's functionalities within Java applications.
- **Performance Optimization:** CFFI allows JMEOS to utilize the performance-optimized code of the MEOS library. C libraries are often used in scenarios where performance is critical, such as handling complex spatiotemporal data and computations. Through CFFI, JMEOS can achieve similar performance levels without rewriting the entire MEOS library in Java.

- **Access to Native Library Features:** Some features and optimizations in the MEOS library might be inherently tied to its implementation in C. CFFI enables JMEOS to access these native features, which might not be directly available or easily replicable in Java.
- **Reducing Redundancy and Maintenance Overhead:** Utilizing CFFI for JMEOS means that the developers can avoid duplicating the functionality already present in the MEOS library. This approach reduces redundancy and simplifies maintenance, as improvements or updates to the MEOS library can be directly utilized in JMEOS without additional reimplementation.

The existing CFFI alternatives for JAVA are the following ones:

JNI

JNI (Java Native Interface) represent a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native applications and libraries written in other languages like C and C++. It is a crucial part of the Java platform, allowing for the integration of native code for performance-critical applications, but requires precise handling to prevent issues related to memory management and platform dependencies.

JNA

JNA (Java Native Access) provides Java programs with direct access to native shared libraries without using JNI. It simplifies the process by automatically handling the conversion of data types between Java and native code. JNA is broadly used for simpler native library integrations and is preferred for its ease of use compared to the more complex and error-prone JNI.

JNR-FFI

JNR (Java Native Runtime) is a lesser-known but modern alternative to JNI and JNA, offering a more simplified approach to calling native libraries from Java. It aims to reduce the overhead and boilerplate code typically associated with JNI, making native integration more accessible and maintainable, especially for complex projects.

Interface	Pros	Cons
JNI	• Full access to native APIs and system libraries. • High performance for complex use cases.	• Complex and error-prone native code handling. • Memory management issues. • Platform-dependent.
JNA	• Simplifies native library access. • Automatic data type conversions. • No native code or JNI boilerplate.	• Lower performance compared to JNI. • Limited support for complex cases.
JNR	• Simplifies integration like JNA. • Better performance than JNA. • Less boilerplate than JNI. • Easier maintenance.	• Lesser community support. • Limited documentation.

Table 2.4: Comparison of JNI, JNA, and JNR

Given the comparison on Table 2.4 between JNI, JNA, and JNR FFI, JNR FFI emerges as the most suitable choice for the development of JMEOS. JNR FFI's efficiency and ease of use align properly with the needs of JMEOS, especially considering its purpose to integrate with MEOS, a foreign library not developed as part of the JMEOS project. This choice mirrors its successful application in projects like JRuby, where JNR FFI has been instrumental in facilitating seamless integration with native libraries.

Opting for JNR-FFI in JMEOS offers significant advantages. Primarily, it enables the development team to stay within the Java ecosystem entirely, without the need to delve into the complexities of writing and maintaining JNI C code. This aspect is particularly beneficial given that MEOS, the native library JMEOS aims to interface with, is an external API, a typical scenario in software development. With JNI, even when not developing the native library oneself, there is still a requirement to write and compile JNI C code, adding to the development overhead.

In contrast, JNR-FFI simplifies the process by eliminating the need for this additional layer of code. Developers can interact directly with the native library using pure Java, which significantly reduces the codebase size compared to a JNI-based approach. This reduction in complexity and code volume not only streamlines the development process but also aligns more closely with the preferences and expertise of Java developers. They can leverage their existing skills and tools within the familiar Java environment, making JNR-FFI an appealing and practical solution for integrating JMEOS with the MEOS library.

2.4 Docker

Docker, a platform built upon containerization technology, represents a paradigm shift in software deployment and development. It allows applications to be packaged along with

their dependencies and runtime environment in containers, ensuring consistency across diverse environments. A Docker container, unlike a virtual machine, is more lightweight as it shares the host system's kernel and prevents the overhead of running a separate operating system instance, as shown in Figure 2.2 . This approach not only enhances portability but also improves resource efficiency and scalability [22].

Docker's containerization strategy is instrumental in managing applications with complex dependencies, like those requiring specific runtime environments. It encapsulates the application's environment, libraries, and settings in a container, ensuring that it functions uniformly regardless of where it is deployed. This consistency addresses the common challenge of discrepancies in software behavior across different systems [32].

In the context of this thesis, Docker occupies a pivotal role due to the following advantages, particularly considering MEOS, a compiled library dependent on machine configuration:

- **Consistent Runtime Environment**: Docker containers encapsulate the specific environment needed by MEOS, ensuring consistent behavior regardless of the host machine's configuration [7].
- **Ease of Deployment**: Docker simplifies the deployment process of JMEOS with MEOS dependencies, allowing for quick and consistent setup across various systems.
- **Isolation and Security**: Containers provide an isolated environment for running JMEOS, minimizing conflicts between diverse applications and enhancing security.
- **Reproducibility**: Docker ensures that the application runs the same way everywhere, which is crucial for testing and debugging MEOS-dependent applications.

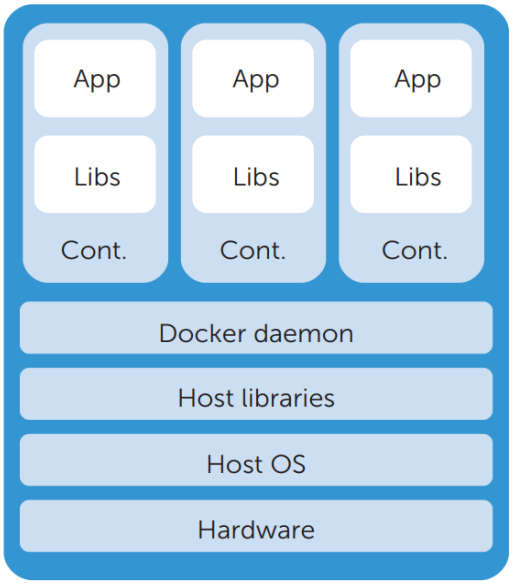

Figure 2.2: Example of a Docker Container [10].

Chapter 3

Design and Implementation

A first JMEOS proof of concept implementation was initially provided by Krishna Chaitanya , three years ago, showcasing its initial feasibility [9]. This implementation utilized JavaCPP FFI [3], a bridging technology that simplifies the integration between Java and native C++ code. JavaCPP stands out for its ability to automatically generate JNI wrappers for C++ libraries, eliminating the need for manual JNI coding. It enables direct translation of C++ API calls to Java methods, streamlining the development process and enhancing code efficiency. JavaCPP is particularly valued for its performance optimization and ease of use, making it a conventional choice for interfacing Java with native code. In this proof of concept, he implemented the `TBox` type constructor and developed several methods that operate on this type, laying the groundwork for further development of JMEOS.

Given that the original proof of concept was developed three years ago and MobilityDB has significantly evolved since then, a decision was made to start anew with the development of JMEOS. This fresh start involves JNR FFI, chosen for its numerous advantages detailed in Chapter 2 of the thesis. The innovative approach also involves deriving inspiration from the global structure of MobilityDB-JDBC, adapting and modifying it to suit the specific needs and capabilities of JMEOS. This strategy aims to ensure that JMEOS aligns closely with the current state and advancements of MobilityDB.

Project Structure

The open-source project MobilityDB-JMEOS is freely accessible on GitHub at the following link: `https://github.com/nmareghn/MobilityDB-JMEOS/tree/main`. The source code of this project is primarily structured into four core components, as shown on Figure 3.1 :

- The first one is the core of the project and includes the design and implementation of various types. This component is located in the `src/main/java` directory and features a range of types and subtypes, such as `Boxes`, `Time`, `Number`, `Text`, `Geo`, and various `Temporal` types.
- The second component comprises a set of functions that facilitate user interaction with the MEOS internal functionalities. This is seamlessly integrated with the core types of JMEOS, offering automatic type conversion provided by JNR FFI. This set of functionalities is organized within the builder and functions directories under `src/main/java`.
- The third component consists of unit tests, essential for verifying the correct functionality of the types and methods in JMEOS. These tests are placed in the test directory and are crucial for ensuring the reliability and stability of the software.

- The final component involves use case benchmarking and visualization, particularly with the BerlinMOD dataset. This aspect of the project is vital for demonstrating the practical application and performance of JMEOS. It is located in the examples directory, providing users with real-world scenarios and examples of how JMEOS can be utilized effectively.

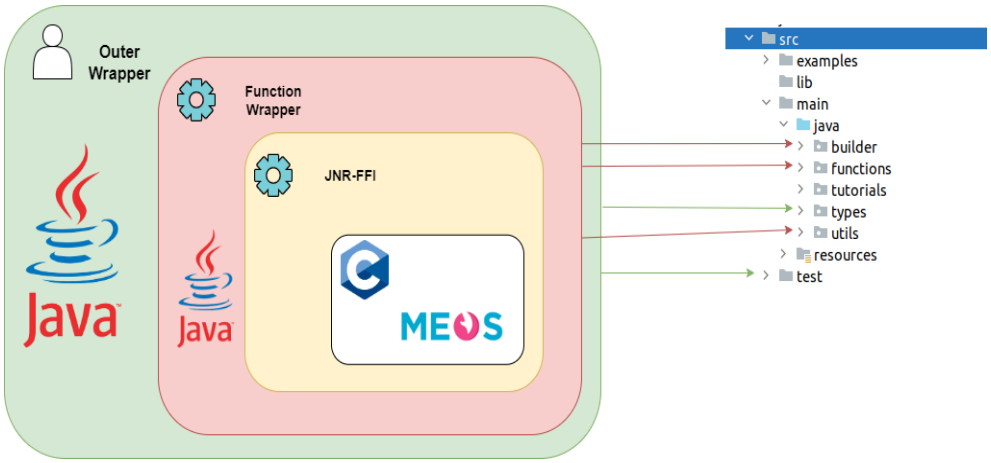

Figure 3.1: JMEOS Package Structure

Structure of the Chapter

This chapter being the core of the thesis, it is crucial to strictly structure it and format it. For each section of this Chapter, the thesis enhances readability by presenting the involved files inside the package tree. This visual representation offers a distinct and structured view of the file organization and hierarchy relevant to each section. Additionally, for the ones focusing on types, an UML (Unified Modeling Language) class diagram is provided. This latter serves as a visual aid to illustrate the relationships between classes and the methods implemented within them. The inclusion of UML diagrams is intended to support the textual description, offering a comprehensive understanding of the class structures and interactions that are detailed in the same section of the thesis. In each diagram arrows describing an implementation of an interface are colored in green, while the direct inheritance is in blue. It is important to pinpoint that no methods were incorporated into the class diagram to avoid having tremendous and not readable figures. More informations about the methods implemented in each class can be found in the Javadoc section or in the Appendix.

The first section will present the JMEOS wrapper that automatically binds MEOS and JMEOS functions. The subsequent sections will present the `Boxes` types as well as the set of `Collections` that includes `base, geo, number, text` and `time` set of types. Then, the abstract `Temporal` types, alongside with the `Instant`, `Sequence` and

`SequenceSet` types will be described in details. The last sections details the main temporal types which are `tbool, ttext, tint, tfloat` and `tpoint`. This structure is purposefully designed for clarity and coherence because boxes types are independent from other types and collections types make usage of `Boxes` while using each others within the collections package. The abstract temporal types being the base of the main temporal types, it is judicious to describe them on first place. For each of the mentioned types, methods will be presented as blocks of groups. Due to the thesis size constraints, groups containing one method will not be detailed. Some groups containing tens of functions, only the most interesting ones will be presented. Throughout the next section the word **inner** will be named, if associated to the description of a method or class, this latter will designate an attribute that stores a corresponding MEOS object. The project can be found in the official github link of MobilityDB: https://github.com/MobilityDB/JMEOS.

3.1 Function Wrapper

The process of wrapping the MEOS library functions remains the key component to implement the rest of the library. The idea behind this project is not to transpose the entire MEOS library source code (written in C) into JAVA code. This would be a waste of time and energy, given that all the classes written in C and linking to MobilityDB would have to be recoded in JAVA. As an alternative, the principle of this work is to take the API, previously integrated with MEOS, and use it to implement JMEOS. In the internal code of the MEOS library, the API file `meos.h` contains all the headers of the functions implemented there.

As a result, wrapping MEOS becomes considerably simpler, since all remaining work is the creation of a set of corresponding function signatures in JAVA, using JNR-FFI. These functions will be called up in any process involving the types to be implemented later. As MobilityDB, and inherently MEOS, is constantly evolving and improving, it is explicit that automatic generation of this wrapper represents the only viable solution for optimal, scalable use of JMEOS.

In the process of automating generation, certain strategic decisions were implemented to enhance efficiency. This procedure is segmented into two main classes, as presented on Figure 3.2 , each with a unique function in the generation process. The first class, named `FunctionsExtractor.java`, is tasked with extracting crucial information from the `meos.h` file. It analyzes the file, pinpointing key lines that contain function signatures and `typedef` declarations. The information gathered is then compiled into two distinct files: `functions.h` and `types.h`.

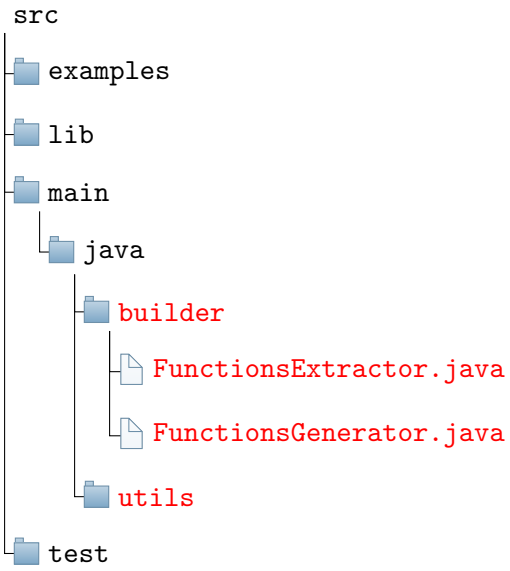

Figure 3.2: Function Wrapper Packages

The second class, known as `FunctionsBuilder.java` is in charge of creating the `functions.java` file. Utilizing the data extracted earlier, this class assembles the contents of the `functions.java` file. It processes the `functions.h` and `types.h` files stored in a temporary folder and constructs the `functions.java` file, which includes the necessary linkages between MEOS's C functions and their Java counterparts. In addition to that, a package named utils and containing utils functions was created. More precisely, this package contains conversion functions, StringBuilder manipulation classes and Serialization functions.

This approach, delineating the generation into two separate classes, enhances the system's modularity and adaptability. It simplifies ongoing maintenance and potential modifications, as each class can be independently adjusted to comply with specific requirements. Furthermore, this automated method of extracting data from the **meos.h** file and generating the **functions.java** file assures precise and consistent bindings between the C and Java functions of MEOS.

3.1.1 Functions Extraction

The class, shown in Figure 3.3, is composed of three Path attributes, one for the input `meos.h` file and two for outputs. It is also composed of two String members used during regex pattern matchings. This class meant to be directly launchable, it contains a `main` function allowing the whole process to run automatically. The idea behind this extraction process is to filter the original input file, to process more easily and faster the generation of functions presented on the next subsection.

The process of functions extraction will iteratively parse the `meos.h` API file and will extract and store the signature of each functions in the file `meos_functions.h`. It will also extract the different structure types contained in the file, and store it in a temporary file named `meos_types.h`.

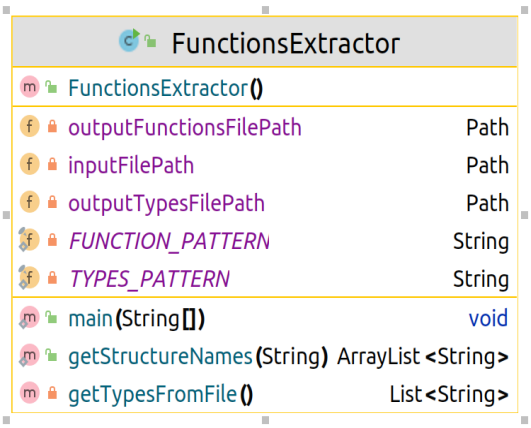

Figure 3.3: Functions Extractor UML Class Diagram

More precisely, a **regex** corresponding to each MEOS functions signature is created. The function extractPatternFromFile from the **BuilderUtils** file contained in the **utils** packages, will generate the corresponding file with the extracted functions signature based on the regex by using **java.util** pattern and matcher methods. Identically, the extraction of the structure types uses the getTypesFromFile and the getStructureNames functions to perform the extraction process, before recording it in the corresponding file through the `BuilderUtils` class.

```
1
2  public static ArrayList<String> extractPatternFromFile(String
       filePath, String regex_pattern) {
3      ArrayList<String> lines = new ArrayList<>();
4      try (BufferedReader reader = new BufferedReader(new
           FileReader(filePath))) {
5          Pattern pattern = Pattern.compile(regex_pattern);
6          String line;
7          while ((line = reader.readLine()) != null) {
8              Matcher matcher = pattern.matcher(line);
9              if (matcher.find()) {
10                 lines.add(matcher.group());
11             }
12         }
13     } catch (IOException e) {
14         e.printStackTrace();
15     }
16     return lines;
17 }
```

Listing 3.1: Extract Pattern From File Function

3.1.2 Functions Generation

The class, shown in Figure 3.4, is composed three `Path` attributes. Two concerns the output obtained from the function extraction process, while the last one will be dedicated for the output of the generation process. The members `equivalentTypes` and `conversionTypes` are respectively dedicated for the types equivalence between the C and Java types, the conversion types between specific C code having an equivalence in Java. Concerning the methods associated to this class, generateFunctions will modify the functions C signature. The functions generateInterface will include the interface nedeed by JNR-FFI into the file. The generateClass method will assemble all pieces to output the final string into the **functions.java** file contained in the **functions** package.

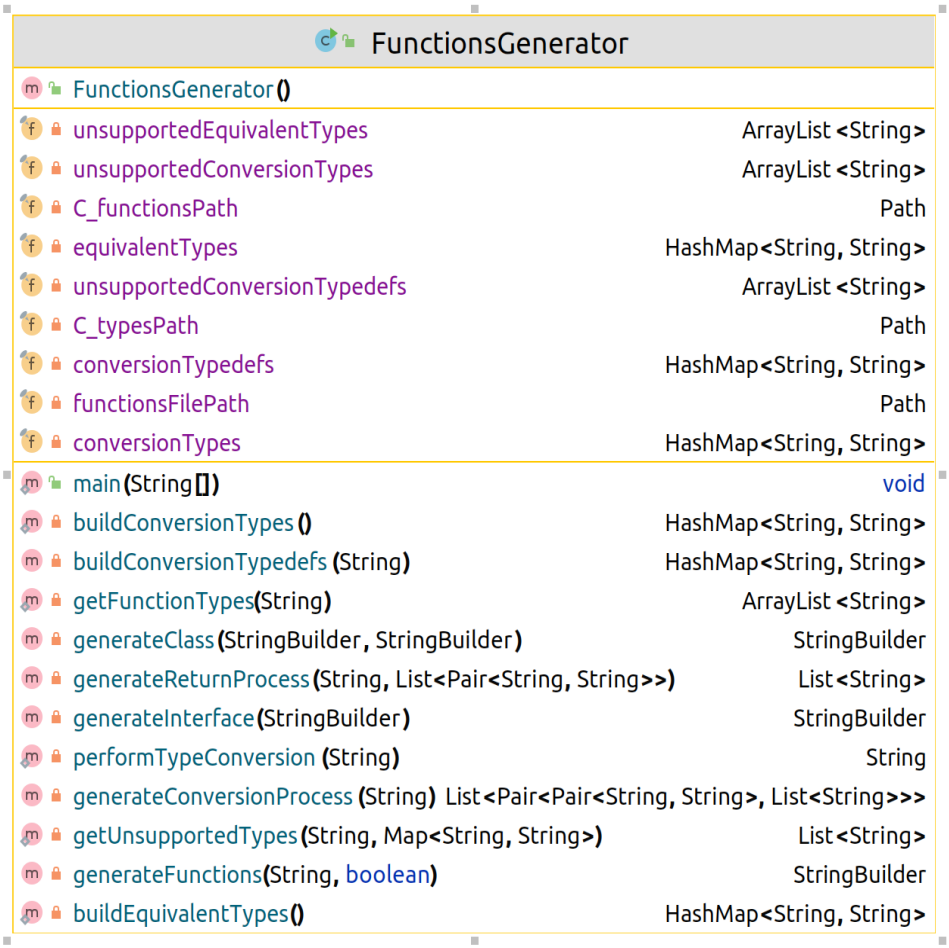

Figure 3.4: Functions Generator UML Class Diagram

The overall process is divided in the three step, previously mentioned:

Generation of Functions Signature

It operates by processing the `meos_functions.h` file, which contains C function signatures only that was previously extracted. Based on the `performTypesConversion` flag, the method decides whether to convert C types into their Java/JNR-FFI equivalents. When this flag is activated, types corresponding to `TimestampTz` and `Timestamp` are respectively converted to `OffsetDateTime` and `LocalDateTime` types coming from **java.time** library. This difference is needed since we divided the interface code, that is called by the inner wrapper, from the Java code that is used in the concrete implementation of JMEOS. This encourages increasing the level of abstraction and to separate the Java code from the JNR-FFI code.

The function starts by creating a StringBuilder obtained from `BuilderUtils` class, which is used to accumulate the Java code for the functions. It was decided to use `StringBuilders` and not a simple `String` because it offers heavier methods to manipulate considerable size of texts.

In addition, the method interprets each line of the file. For each signature, it performs a type conversion process, adapting C syntax and types for Java, as displayed on Table 3.1 . This includes handling pointers, removing C-specific keywords, and adjusting function parameters.

Additionally, it uses `conversionTypedefs` to handle typedef conversions in C, ensuring the Java equivalents are accurately reflected.

The method is diligent in checking for unsupported types by comparing them against `equivalentTypes`, `conversionTypes`, and `conversionTypedefs`. After processing each line, the transformed Java function declaration is appended to the `StringBuilder`. The ultimate output of the method is this `StringBuilder`, filled with Java declarations of the C functions, now adjusted for Java compatibility.

In summary, `generateFunctions` methodically transforms C function signatures into a format suitable for Java, enabling the creation of Java files that serve as an interface to the MEOS C library functions. This process is vital in bridging C libraries with Java applications, particularly for the integration of MEOS functionalities.

C Type	**JNR-FFI/Java Type**
`Timestamp`	`LocalDateTime`
`TimestampTz`	`OffsetDateTime`
`*`	`Pointer`
`*char`	`String`
`**char`	`Pointer`
`bool`	`boolean`
`float`	`float`
`double`	`double`
`void`	`void`
`int`	`int`
`short`	`short`
`long`	`long`
`int8`	`byte`
`int16`	`short`
`int32`	`int`
`int64`	`long`

Table 3.1: Correspondence of Types Between C and JNR-FFI/Java

Generation of Interface

The generateInterface method constructs the Java interface for the MEOS library. This method takes a `StringBuilder` named `functionsBuilder`, which contains the Java declarations of the C functions that have been processed and converted.

The method initiates by creating a new `StringBuilder` object, builder, to construct the interface. It starts appending to builder the fundamental structure of a Java interface named MeosLibrary. This includes the declaration of the interface itself and the initialization of an instance of the `MeosLibrary` class, achieved using the `JarLibraryLoader` create method. This design pattern ensures that the MeosLibrary interface can be easily accessed and used throughout the Java code.

Following this setup, the method incorporates the function declarations from the `functionsBuilder` into the builder. This integration is performed by appending each function declaration, ensuring that the entire collection of converted C functions becomes part of the MeosLibrary interface.

Generation of Class

The generateClass function combines the outputs of the generateInterface and generateFunctions processes to create a final functional Java class.

The method receives two inputs: `functionsBuilder` and `interfaceBuilder`, both obtained as an output of the previous generation steps. Collectively, these inputs provide the foundational elements needed for the class creation.

The function begins by establishing a new `StringBuilder`. This builder is the backbone for constructing the Java class, where it methodically appends the necessary class structure elements, including package declarations and import statements. The return process is added to each declaration of Java functions. Depending on the parameters, some functions require a specific handling. Some functions return a new JNR-FFI `Pointer` based on the return value of the interfaced primary function. This `Pointer` is obtained by managing the runtime system and allocating a memory space in this runtime system, as shown on the Listing 3.2.

```
1    functionCallingProcess.add("Pointer result =
        Memory.allocateDirect(runtime, Long.BYTES);");
2    functionCallingProcess.add("out = MeosLibrary.meos." +
3            BuilderUtils.extractFunctionName(signature) + "(" +
                BuilderUtils.getListWithoutBrackets(paramNames) +
                ");");
4    functionCallingProcess.add("Pointer new_result =
        result.getPointer(0);");
5    functionCallingProcess.add("return out ? new_result : null
        ;");
```

Listing 3.2: Part of the FunctionsGeneration File

Finally, the methods concatenate all string builders in one and returns it. The returned string builder is written into the `functions.java` class, contained in the

functions package, as seen in Listing 3.3.

```
1
2  public class functions {
3      public interface MeosLibrary {
4          MeosLibrary INSTANCE =
               JarLibraryLoader.create(MeosLibrary.class,
               "meos").getLibraryInstance();
5          MeosLibrary meos = MeosLibrary.INSTANCE;
6          Pointer lwpoint_make(int srid, int hasz, int hasm,
               Pointer p);
7          Pointer lwgeom_from_gserialized(Pointer g);
8          Pointer gserialized_from_lwgeom(Pointer geom, Pointer
               size);
9          int lwgeom_get_srid(Pointer geom);
10         double lwpoint_get_x(Pointer point);
11         double lwpoint_get_y(Pointer point);
12         ...
```

Listing 3.3: Part of the Functions Output File

```
1          public static Pointer lwpoint_make(int srid, int hasz,
               int hasm, Pointer p) {
2          return MeosLibrary.meos.lwpoint_make(srid, hasz, hasm, p);
3         }
4          public static Pointer lwgeom_from_gserialized(Pointer g) {
5              return MeosLibrary.meos.lwgeom_from_gserialized(g);
6          }
7          public static Pointer gserialized_from_lwgeom(Pointer
               geom, Pointer size) {
8              return MeosLibrary.meos.gserialized_from_lwgeom(geom,
                   size);
9          }
10         public static int lwgeom_get_srid(Pointer geom) {
11             return MeosLibrary.meos.lwgeom_get_srid(geom);
12         }
13         public static double lwpoint_get_x(Pointer point) {
14             return MeosLibrary.meos.lwpoint_get_x(point);
15         }
16         public static double lwpoint_get_y(Pointer point) {
17             return MeosLibrary.meos.lwpoint_get_y(point);
18         }
19         ...
```

Listing 3.4: Part of the Functions Output File Continuation

3.1.3 Utility Functions

Contained in the package **utils**, utils classes are implemented to enhance JMEOS capabilities. `BuilderUtils` is a class that was previously used during the whole wrapping

of functions process. All `StringBuilder` operations, such as concatenation or file writing are defined in this class. Moreover, this class was crucial for formatting correctly the `StringBuilders`, for extracting the parameters names, types and functions signature that was used during the extraction process. The other important utility class is the `ConversionUtils` class that define conversion functions between MEOS and Java types. These functions remain the exclusive ones that are manually defined and used in the types manipulation of JMEOS. In this class, conversions between `TimestampTz` and `LocalDateTime` and vice versa can be found. We can equally find conversions between `Gserialized` and `Geometry/Point` (from the **jts** package).

3.2 Boxes

A bounding box represents the smallest box that completely encases a temporal object in either 1D, 2D, or 4D space, depending on the object's nature. The **boxes** package is located under the **types** package of the project, as shown on Figure 3.5.

There are two different type of boxes that will be implemented in accordance to MEOS types:

- `Period`: This type is not directly a bounding boxes but is employed for `tbool` and ttext types, focusing solely on the temporal extent, resulting in a 1D box.
- `Tbox` (temporal): Applicable to tint and tfloat types, this bounding box records the value extent along the X-axis and the temporal extent on the T-axis, forming a 2D box.
- `STbox` (spatiotemporal): Designed for `tgeompoint` and `tgeogpoint` types, the `STbox` encapsulates the spatial extent across the X, Y, and Z dimensions, while the temporal extent is mapped to the T dimension, resulting to a 4D box.

Bounding boxes are useful when manipulating temporal objects under indexing operations for efficiency reasons.

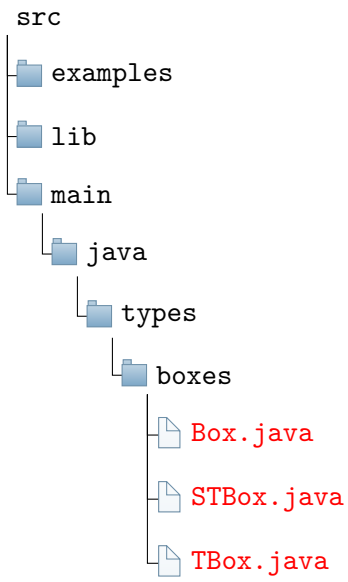

Figure 3.5: Boxes Package Structure

3.2.1 TBox

`TBox` encapsulate the range of values that a temporal element, like time intervals or varying numerical values over time, can occupy. For example, a `TBox` can be defined as follow: `TBOXINT XT([0, 10),[2023-06-01, 2023-06-05])` representing an integer `tbox`. This section will present some functions and operators grouped by categories. For simplicity and due to the thesis size constraints, only three to four operators per section will be detailed. In JMEOS, the `TBox` type is implementing the `boxes` interface, which is itself implementing the `TemporalObject` interface, depicted in Figure 3.6 . Interfaces in Java are crucial as they allow for the definition of a contract that classes can implement, promoting a form of abstraction and enabling multiple inheritance of type. This functionality amplifies flexibility and scalability in software design [8]. This construction will enable us to re-create a polymorphic functions designs similar to MobilityDB.

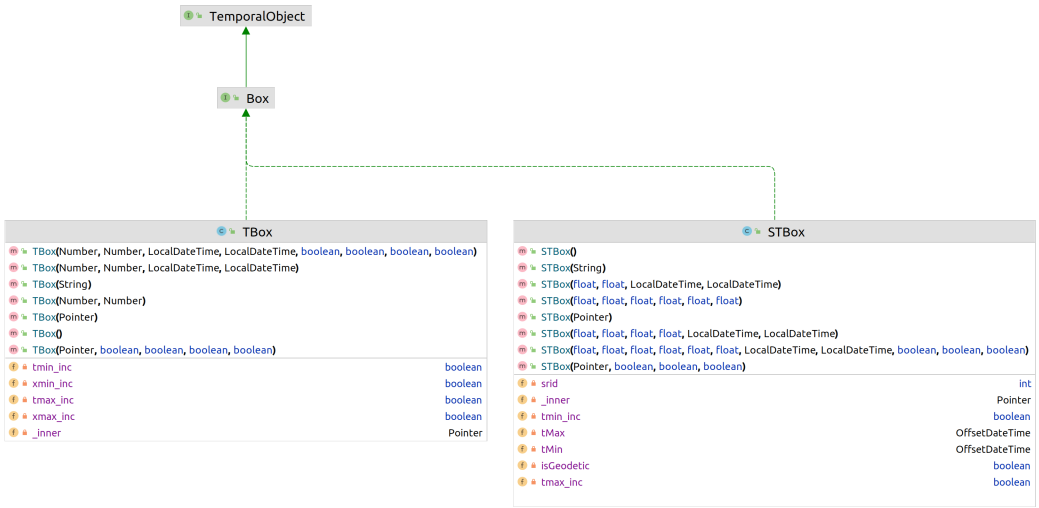

Figure 3.6: Boxes Class Diagram Structure

Constructors

Seven different constructors are implemented. In order to rigourously transpose PyMEOS constructors of this class in Java, it should have contained more than 1,100 different constructors. It is clear that such number of constructors is not viable in terms of maintainability or readability. This number is due to the fact that Python authorized optional argument by default since its first version [40]. Java does not support optional or default parameters in method signatures, emphasizing clear and explicit method contracts. This design choice avoids the complexity and potential ambiguity associated with parameter defaults, ensuring more predictable method behaviors [18]. Thus, it was decided to minimize the number of constructors by keeping them less flexible and let the user explicitly input the parameters value.

1. The first interesting constructor is the string constructor. It accepts a string, representing the `TBox` in MobilityDB fashion, as the only parameter.

```
1    TBox tbox = new TBox("TBOXFLOAT XT([1, 2],[2019-09-01,
         2019-09-02]");
```

Listing 3.5: String Constructor TBox

2. The second one takes as parameter the minimal and maximal x coordinates but also the minimal and maximal `LocalDateTime` as well as the inclusion of these four parameters as a boolean predicate.

```
1    TBox tbox = new TBox(1, 2, dt1, dt2, true, false, true,
         false);
```

Listing 3.6: Full Constructor TBox

3. The third constructor that is presented in this section, is the `Pointer` constructor. It allows a user to create a `TBox` object from the MEOS corresponding object. This returned objected is stored in the `_inner` field 3.6 of the `TBox` type.

```
TBox tbox = new TBox(p);
```

Listing 3.7: Pointer Constructor TBox

4. The last constructor,from_time, allows to create a `TBox` object from a `Period`, `PeriodSet` or `TimestampSet`.

```
Period period = new Period("[2019-09-01, 2019-09-02]");
TBox t = TBox.from_time(period);
//"TBOX T([2019-09-01 00:00:00+00, 2019-09-02
   00:00:00+00])"
```

Listing 3.8: From Time Constructor TBox

Output

1. Output method allows users to visualize the string representation of a `TBox`.The unique method performing this action is the toString method that implicitly uses the tbox_out method of MEOS.

```
tbox.toString(15);
//"TBOXFLOAT XT([1, 2],[2019-09-01 00:00:00+00,
   2019-09-02 00:00:00+00])"
```

Listing 3.9: Output Method TBox

Conversions

1. Conversions methods permit to transform a `TBox` into a `period` or a `floatspan`.

```
TBox tbox = new TBox("TBOXFLOAT X([1,2])");
tbox.to_period(); //Period("[2019-09-08 02:03:00+0,
   2019-09-10 02:03:00+0]")
tbox.to_floatspan(); //FloatSpan(1.0f, 2.0f, true, true)
```

Listing 3.10: Conversions Method TBox

Topological Operations

1. Topological methods allow the user to perform specific topological operations between `TBox` and `FloatSpan`, `TNumber` or `TBox`. These methods return a boolean and examine the adjacency, the capacity, the overlapping and the identity.

```
TBox tbox = new TBox("TBOXFLOAT X([1,2])");
TFloatInst tf = new TFloatInst("4.5@2019-09-01");
tbox.is_adjacent_tbox(tf); //false
tbox.contains(tbox); //true
tbox.overlaps(tf); //false
tbox.is_same(tbox); //true
```

Listing 3.11: Topological Operations TBox

Position Operations

1. Similarly to topological operations, positions methods permits to test the position of `TBox` against another `TBox` or a `TNumber` (temporal integer or float). Eight methods are implemented in total, testing for with left, right, after or before operators.

```
TBox tbox = new TBox("TBOXFLOAT X([1,2])");
TFloatInst tf = new TFloatInst("4.5@2019-09-01");
tbox.is_left(tf);
tbox.is_right(tbox);
tbox.is_after(tf);
tbox.is_before(tbox);
```

Listing 3.12: Position Operations TBox

Set Operations

1. Set operations performs set methods between two `TBox`. Users can perform `intersection` or `union` between two `TBox` and obtain a new one.

```
TBox tbox = new TBox("TBOXFLOAT X([1,2])");
tbox.intersection(tbox);
tbox.union(tbox);
```

Listing 3.13: Set Operations TBox

Comparisons

1. The last group of methods are comparisons operations. One can compare two `TBox` for equality, inequality, superiority or inferiority. Since Java does not accepts operators overloading, i.e. overload an operator such as = between two customs type, it was decided to create manually defined functions names.

```
TBox tbox = new TBox("TBOXFLOAT X([1,2])");
tbox.eq(tbox);
tbox.notEquals(tbox);
tbox.greaterThan(tbox);
```

Listing 3.14: Comparisons Operations TBox

3.2.2 STBox

`STBox` or Spatiotemporal represent the spatial and temporal extents of an object. It defines boundaries in both spatial (X, Y, Z) and temporal (T) dimensions, encapsulating where and when an object exists or an event occurs within a spatiotemporal context [17]. For example, a `STBox` can be defined as follow: `STBOX ZT(((1, 1, 1),(2, 2, 2)),[2019-09-01,2019-09-02])` representing an integer `tbox`.

Similarly to `TBox`, the section will present some functions and operators grouped by categories. In JMEOS, the `STBox` type is implementing the boxes interface, which is itself implementing the `TemporalObject` interface, depicted in Figure 3.6.

Constructors

Eight different constructors are implemented. Again, similarly to `TBox`, this number of methods is intentional to prevent tremendous number of constructors. Therefore, it was decided to minimize it by keeping them less flexible and permit the user explicitly input the parameters value.

1. The first interesting constructor is the string constructor. This constructor is the simplest one and it needs to be favorised when constructing a `STBox` object to avoid as much as possible any type or compatibility errors. It accepts a string, representing the `STBox` in MobilityDB fashion, as the only parameter. The string contains the tri-dimensional coordinates as well as the temporal dimension.

```
1    STBox stbox = new STBox("STBOX ZT(((1, 1, 1),(2, 2,
        2)),[2019-09-01,2019-09-02])");
```

Listing 3.15: String Constructor STBox

2. The second one considers as parameter a float representing; the minimal and maximal x coordinates, the minimal and maximal y coordinates, the minimal and maximal z coordinates but also the minimal and maximal `LocalDateTime` as well as the inclusion of this last parameter. The particularity of this constructor is that the temporal dimension is encompassed into a `Period` object and passed as an argument to the MEOS internal constructor.

```
1    STBox stbox = new STBox(1.0, 2.0, 1.0, 2.0, 1.0, 2.0,
        dt1, dt2, true, true);
```

Listing 3.16: Full Constructor STBox

3. The third constructor that is presented in this section, is the `Pointer` constructor. It allows users to create a `STBox` object from the MEOS corresponding object. This returned objected is stored in the `_inner` field 3.6 of the `STBox` type.

```
STBox stbox = new STBox(p);
```

Listing 3.17: Pointer Constructor STBox

4. The last constructor,`from_geometry`, allows to create a `TBox` object from a `Geometry` object, obtained from the **jts** library, combined with a `Period` or a `LocalDateTime`. This function implicitly use **Utils** conversion functions to convert the `geometry` into a `gserialized`.

```
Period period = new Period("[2019-09-01, 2019-09-02]");
Geometry geom = new Geometry();
STBox st = STBox.from_geometry(geom,period);
```

Listing 3.18: From Geometry Constructor STBox

Output

1. Output method allows the user to visualize the string representation of a `STBox`.The unique method performing this action is the toString method that implicitly uses the tbox_out method of MEOS. It implements the maximal number of decimals as parameter.

```
stbox.toString(15);
//"STBOX XT(((1, 1),(2, 2)),[2019-09-01,2019-09-02])"
```

Listing 3.19: Output Method STBox

Conversions

1. Conversions methods permit to transform a `STBox` into a `period` or a `Geometry` object. This latter returns the spatial dimension of the `STBox`.

```
STBox stbox = new STBox("STBOX XT(((1, 1),(2,
    2)),[2019-09-01,2019-09-02])");
stbox.to_period();
stbox.to_geometry() ;
```

Listing 3.20: Conversions Operations STBox

Transformations

1. Transformation methods are able to modify directly the `STBox` object. Typically, ones would rather only work the spatial dimension of the object or expand it with spatial references. These operations are possible thanks to `get_space` or `expand` methods.

```
1    STBox stbox = new STBox("STBOX XT(((1, 1),(2,
        2)),[2019-09-01,2019-09-02])");
2    STBox new_st = stbox.get_space();
3    STBox sec_st = stbox.expand(1.0);
```

Listing 3.21: Transformations Operations STBox

Position Operations

1. Similarly to `TBox` operations, positions methods permits to test the position of `TBox` against an other `TBox` or a `TNumber` (temporal integer or float). Eight methods are implemented in total, testing for with left, right, after or before operators. The difference with `TBox`, is the fact that the parameter can be any spatiotemporal object such as a `Period`, a `Temporal` or a `Geometry` object.

```
1    STBox stbox = new STBox("STBOX XT(((1, 1),(2,
        2)),[2019-09-01,2019-09-02])");
2    TFloatInst tf = new TFloatInst("4.5@2019-09-01");
3    stbox.is_left(tf);
4    stbox.is_right(stbox);
```

Listing 3.22: Position Operations STBox

Set Operations

1. Set operation performs set methods between two `STBox`. Users can perform intersection or unions between two `STBox` and obtain a new one.

```
1    STBox stbox = new STBox("STBOX XT(((1, 1),(2,
        2)),[2019-09-01,2019-09-02])");
2    stbox.intersection(stbox);
3    stbox.union(stbox);
```

Listing 3.23: Set Operations STBox

Comparisons

1. One can compare two `STBox` for equality, inequality, superiority or inferiority. Since Java does not accepts operators overloading, i.e. overload an operator such as = between two customs type, it was decided to create manually defined functions names.

```
1    STBox stbox = new STBox("STBOX XT(((1, 1),(2,
         2)),[2019-09-01,2019-09-02])");
2    stbox.eq(stbox)
3    stbox.notEquals(stbox)
4    stbox.greaterThan(stbox)
5    stbox.lessThan(stbox)
```

Listing 3.24: Comparisons Operations STBox

3.3 Collection

In the context of JMEOS, the **Collection** package contains various base types named `Span`, `SpanSet`, and `Set`. It also contains inherited types from these three abstract base types, such as `text`,`geo` or numbers, as shown on Figure 4.1.

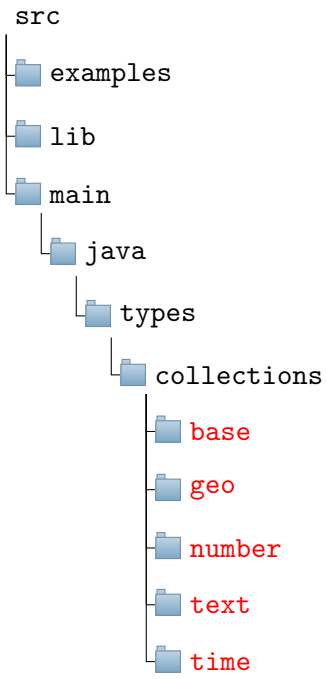

Figure 3.7: Collection Package Structure

3.3.1 Base

The structure of the three base type classes are depicted on the Figure 3.8.

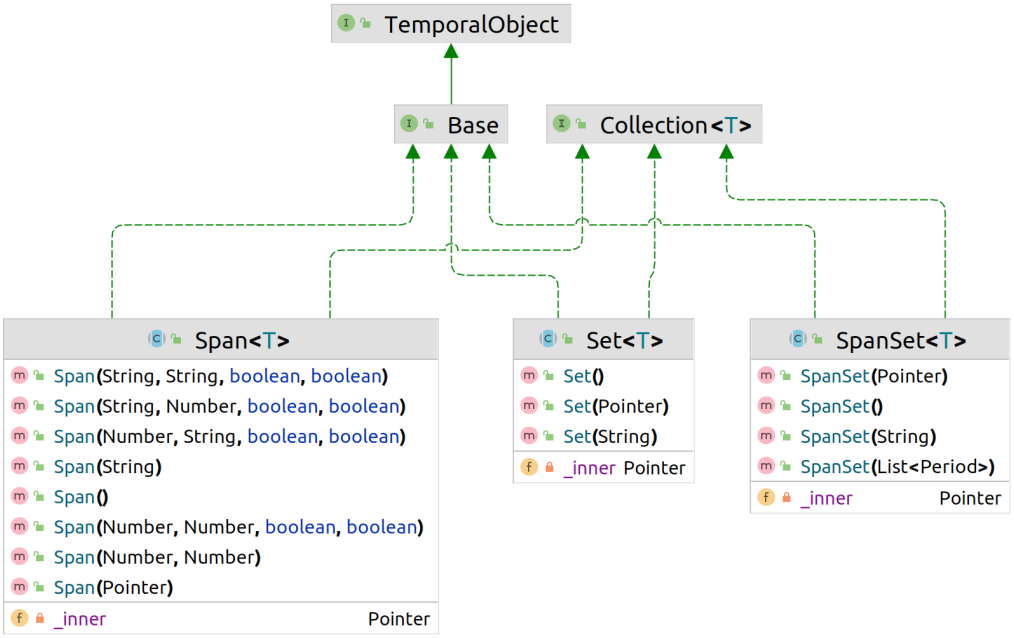

Figure 3.8: Base Classes Diagram Structure

3.3.1.1 Set

In MobilityDB, `set` types are similar to one-dimensional array types in PostgreSQL but with a constraint that they do not contain duplicates. These set types can be based on various base types like `integer`, `float`, `text`, as well as spatial types like `geometry` and `geography` from PostGIS [42].

In JMEOS, the abstract `Set` type will act as the parent type for `GeoSet`, `FloatSet`, `IntSet`, `TextSet` and `TimestampSet`. This type implements `Base` interface to outline its ordinary behavior with the two other base types, and `Collection` interface that will be needed later on the `Time` types implementation.

Constructor

As for the previous section describing `Box` types, this subsection will provide a brief explanation of the two important constructor implementations.

1. Implementation of a constructor on an abstract class is counter intuitive since such class are never meant to be instantiated. Despite this fact, many methods are implemented inside this abstract class and since these methods needs a viable

(containing inner MEOS object) object to be called, it is necessary to produce a sketch to apply these methods. With PyMEOS, it is straightforward since it is possible to directly call child class from its parent abstract class. In JMEOS, it was decided to create abstract methods (implemented by child class) which return value will be stored into the inner MEOS variable of the abstract class. This little sketch will be used multiples times in many abstract classes throughout the project. The first constructor is based on a String parameter and will subsequently call the child constructor that accepts a string as parameter and which returns the child inner `Pointer`.

```
1     public Set(String str){
2         this._inner = createStringInner(str);
3     }
4     public abstract Pointer createStringInner(String str);
```

Listing 3.25: String Constructor for Set

2. The second constructor is similar to the first one, in terms of principle, the only difference resides in its parameter which is a `Pointer` parameter rather than a `String` one.

```
1     public Set(Pointer inner){
2         this._inner = createInner(inner);
3     }
4     public abstract Pointer createInner(Pointer inner);
```

Listing 3.26: Pointer Constructor for Set

Topological Operations

- Three topological operations are implemented for the `Set` type: is_contained_in, contains and overlaps. As its name suggests, it allows the users to check if a `Set` is contained, contains or overlaps another one. These three functions taking a `Base` type as parameter, it throws an error if this latter does not correspond to a `Set`.

Position Operations

- Four position operations can be performed by any type inheriting from the `Set` type. One can evaluate if a `Set` is strictly to the left (is_left) of an other `Set` meaning that the first one strictly ends before the second starts. It is also possible to check if a Set is strictly to the right (is_right)or if it is overright/overleft with the methods (is_over_or_right, is_over_or_left) , these latter allowing for overlaps.

Distance Operations

- The only distance operator named distance allows users to compute the distance between sets other sets but also `spans` and `SpanSet`. The returned distance is

expressed in `float` java type.

Comparisons

- Analogous to the past types implemented, ones can compare sets between them for equality, inequality, superiority or inferiority.

3.3.1.2 Span

`Span` types in MobilityDB are analogous to the range types in PostgreSQL but with specific constraints. They are of determined length, do not allow empty spans or infinite bounds, and are designed for increased performance by removing the overhead of processing variable-length types. Examples of `span` types include `intspan`, `floatspan`, and `tstzspan` [42].

The abstract `Span` is the parent type for `FloatSpan`, `IntSpan` and `Period`. This type implements `Base` interface to outline its common behavior with the two other base types, and **Collection** interface that will be needed later on the `Time` types implementation.

Constructor

Compared to the `Set` constructors, the `Span` type comprises 8 different constructors:

1. The first interesting constructor to describe, is clearly, the string constructor which takes a `String` as a parameter and return the inner MEOS `Pointer` of the child class.

```
1    public Span(String str){
2        this._inner = createStringInner(str);
3    }
4    public abstract Pointer createStringInner(String str);
```

Listing 3.27: String Constructor for Span

2. The second ones are all `Number` constructors involved. Initially, the child constructor that is associated to this subset of constructors takes as parameter two numbers (float or integer) as well as two booleans that indicates if the bounds of the `Span`. For more flexibility, five different constructors were implemented providing the possibility to the user to write strings or numbers and also to omit writing the corresponding bounds.

```
1     public Span(java.lang.Number lower, java.lang.Number
          upper, boolean lower_inc, boolean upper_inc){
2         this._inner = createIntInt(lower, upper, lower_inc,
              upper_inc);
3     }
4     public Span(java.lang.Number lower, String upper, boolean
          lower_inc, boolean upper_inc){
5         this._inner = createIntStr(lower, upper, lower_inc,
              upper_inc);
6     }
7     public Span(String lower, String upper, boolean
          lower_inc, boolean upper_inc){
8         this._inner = createStrStr(lower, upper, lower_inc,
              upper_inc);
9     }
10    public Span(String lower, java.lang.Number upper, boolean
          lower_inc, boolean upper_inc){
11        this._inner = createStrInt(lower,upper, lower_inc,
              upper_inc);
12    }
13    public Span(java.lang.Number lower, java.lang.Number
          upper){
14        this._inner = createIntIntNb(lower, upper);
15    }
```

Listing 3.28: Full Constructor for Span

3. The last constructor is exactly similar to the `Pointer` constructor of the `Set` type. Thus, it will not be described or technically detailed.

Topological Operations

- Five topological operations are implemented for the `Span` which are is_adjacent, is_contained_in, contains, is_same and overlaps. These methods are similar to the ones presented previously. One particularity is the fact that all five methods accept operations with a `SpanSet` and not only with `Span` type.

Position Operations

- `Span` abstract type contains four position operations which are the same as the one described in the `Set` type. The difference, here, resides in the possibility that `Span` type can be positionally compared to `spans` and `spanset`.

Distance and Comparisons operations

- Both groups of operators are similar to the `Set` type but adapted to `Span`.

Set Operations

- Set operations are implemented, providing two methods. The first one allow a user to perform intersection between two spans or between a `span` and a `spanset`. The second one permits the union between similar types of parameters.

3.3.1.3 SpanSet

These are similar to `span` types but correspond more closely to multirange types in PostgreSQL. They inherit the constraints of `span` types and are used to represent a set of `spans`. Examples include `intspanset`, `bigintspanset`, `floatspanset`, and `tstzspanset` [42].

`SpanSet` is the parent type for `FloatSpanSet`, `IntSpanSet` and `PeriodSet`. As for the two last abstract type presented, it implements Base interface for the similar reasons.

Constructor

Compared to the `Set` constructors, the `Span` type comprises 8 different constructors:

1. As for the two last presented types, `SpanSet` type contains `String` and `Pointer` constructors that works exactly in the same fashion than the other `Base` types.

2. The third constructor is more specific to `SpanSet` , in the sense that it takes a list of `Periods` as parameter. This third constructor was built to fit with the `PeriodSet` constructor specificity giving more flexibility to users.

```
1     public SpanSet(List<Period> periods){this._inner =
          createListInner(periods);}
```

Listing 3.29: List Constructor for SpanSet

Other Operations

- All others operations are exactly similar to the `Span` operators in the exception that is dedicated to `SpanSet` and so using MEOS `spanset` functions. Thus, to avoid any text duplication, it was decided to not present them in this section.

3.3.2 GeoSet

The `GeoSet` type is directly inherited from the base type `Set`. It is a generic abstraction that encapsulates a the `GeographySet` and `GeometrySet` types. `GeographySet` is designed for geodetic (earth-based) coordinates and supports global, round-earth operations, while `GeometrySet` is used for planar or Cartesian coordinates suitable for localized spatial operations. The exact class structure is depicted on Figure 3.9.

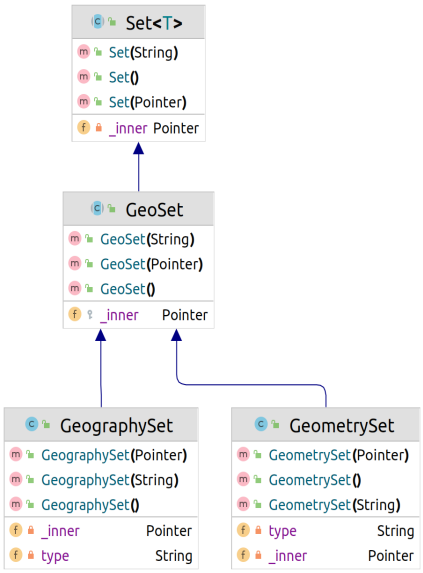

Figure 3.9: GeoSet Class Structure

Both concrete classes do not contain any specific methods. It exclusively contains one `String` constructor and one `Pointer` constructor similarly implemented to the rest of the work. The behavior of these classes is defined in the `GeoSet` abstract class.

Constructor

1. As for the other presented types of this section, a String and a Pointer constructor were implemented allowing users to easily create Geography/Geometry sets.

2. Moreover, a factory method was created, in order to establish the distinction between geographic and geometry sets for methods that requires it, due to the fact that the behavior of the concrete classes are described in the parent one.

```
1     public static GeoSet factory(String type, Pointer inner){
2         if (type == "Geom"){
3             return new GeometrySet(inner);
4         } else if (type == "Geog") {
5             return new GeographySet(inner);
6         }
7         return null;
8     }
```

Listing 3.30: Factory Method for GeoSet

Output

- Three output methods permits to visually observe the content of a `GeoSet` object. The first one named as_wkt provide a Well-known text representation of a `GeoSet`. The second one as_ewkt output an Extended Well-known text representation of the object, while the last one toString provide a simple String description of the inner object.

Accessors

- The start_element method returns the first element of the set, marking the beginning of the spatial sequence, while the end_element method retrieves the last element, indicating the end of the sequence. Additionally, considering we are dealing with geometric data, an accessor method for the SRID (Spatial Reference System Identifier) has additionally been included, allowing for the retrieval of the SRID value associated with the geometries in the set.

Set Operations

- The union, intersection and subtraction between `GeoSets` and `Geometry` are made possible with the respective implemented methods.

3.3.3 Number

The Number package in JMEOS groups together diverse classes that handle numeric temporal data: `FloatSet`, `IntSet`, `FloatSpan`, `IntSpan`, `FloatSpanSet` and `IntSpanSet`. Each inheriting from the abstract classes `Span`, `SpanSet`, and `Set`. The following subsections will present float and integer inherited types as a pair since they share in common the same methods behaviors, dedicated to the base type they represent. The classes representation is depicted on Figure 3.10.

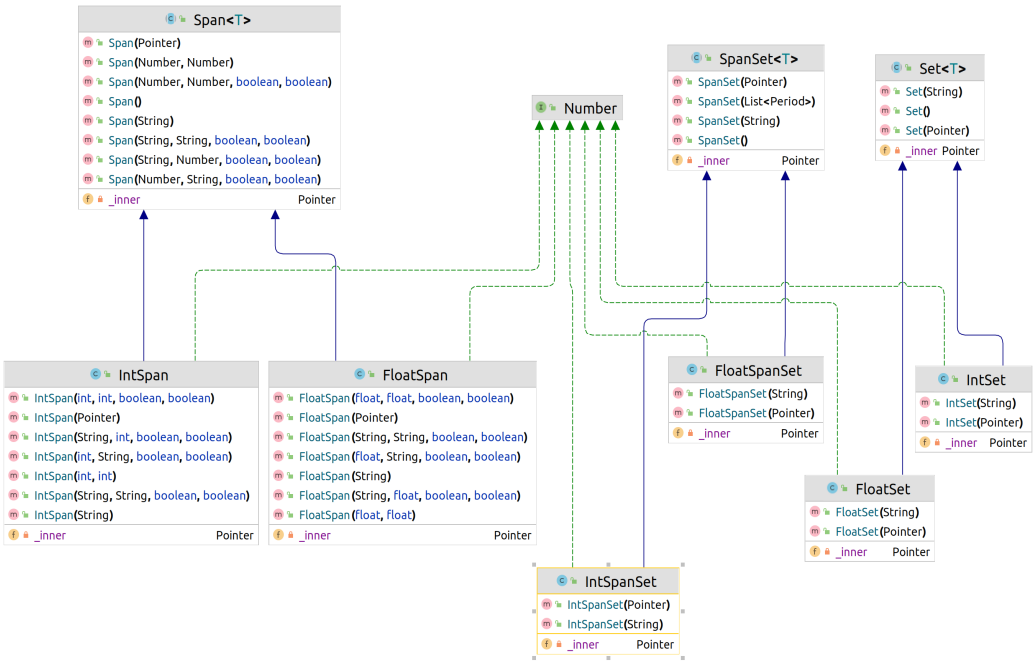

Figure 3.10: Number Class Structure

3.3.3.1 FloatSet and IntSet

`FloatSet` is a set of distinct floating-point values that occur at specific times, similar to an array but without duplicate values. Similar to `FloatSet`, IntSpan class holds a set of distinct integer values that occur at discrete times, ensuring no repetitions.

Constructor

1. The constructors for both classes are the same as the ones depicted for their parent abstract class `Set`. The only difference between them is the base type of the constructor's parameters which are floats for `FloatSet` and integer for `IntSet`. Thus it is unnecessary to present them or detail them in this section.

Conversions

- Typically, a `FloatSet`/`IntSet` can be converted into a `FloatSpanSet`/`IntSpanSet` new object through the method `to_spanset`. Analogously, it can also be converted into a `Span` thanks to the method `to_span`.

```
1    FloatSet fs = new FloatSet("{1, 3, 56}")
2    FloatSpanSet fsps = fs.to_spanset();
3    FloatSpan fsp = fs.to_span();
```

Listing 3.31: Conversion Operations for FloatSet

Accessors

- The start_element method returns the first element of the set, marking the beginning of the spatial sequence, while the end_element method retrieves the last element, indicating the end of the sequence. Additionally, the element_n return the nth element of the `FloatSet/IntSet`.

Transformations

- Concerning transformations operations, ones can perform a shift, of a certain value, of all the elements present in the set through the shift(float delta) functions. It is also possible to scale the elements inside the set with a certain width thanks to the scale method. To do both operations within a `FloatSet/IntSet`, the shift_scale operations was made available.

Position Operations

- Four position operations, corresponding to is_left, is_right, is_over_or_right and is_over_or_left, respectively verify if a float/integer or a `Base` type is positioned strictly on the left, right, right overlapping or left overlapping. A code snippet describing one position operator is provided below:

```
1    FloatSet fs = new FloatSet("{5, 10}");
2    FloatSet fs2 = new FloatSet("{1, 2, 3}");
3    fs2.floatset.is_left(fs); //true
```

Listing 3.32: Position Operations for FloatSet

Set Operations

- The last group of operations performs `union`, `intersection` and `subtraction` between two `FloatSet/IntSet` or between one of this latter set and its base type, i.e. `integer` or `float`.

3.3.3.2 FloatSpan and IntSpan

`FloatSpan` represents a continuous range of floating-point values, encapsulating the start and end of numeric values, generally in a temporal context. Analogous to `FloatSpan`, IntSpan defines a continuous range of integer values, in the same fashion.

Constructor

1. Constructors for both classes are the same as the ones described for their parent abstract class `Span`. The only difference between them is the fact that among the multiple constructors taking as parameter a number, `FloatSpan` will only accept float values, while `IntSpan` only integer values. Thus, it is not necessary to present them or detail them in this section.

Conversions

- Typically, a `FloatSpan`/`IntSpan` can be converted into a FloatSpanSet/IntSpanSet new object through the method to_spanset. Analogously, a `FloatSpan` can be converted to a new IntSpan object and vice-versa thanks to the to_intspan/tofloatspan methods.

```
1     IntSpan intspan = new IntSpan("[7, 10)");
2     FloatSpan sp = intspan.tofloatspan();
```

Listing 3.33: Conversion Method for IntSpan

Accessors

- The implementation of both classes gives an access to the lower, upper and width value respectively through the lower, upper and width methods.

Transformations

- Transformations operators are exactly similar to the ones described in the past section, having the same behavior.

Topological Operations

- Three topological operations are implemented for each class. The adjacency between a float or an integer is performed thanks to the is_adjacent functions. The capacity and identity operations are respectively processed by contains and is_same method. A code snippet describing one topological operator is provided below:

```
1     IntSpan intspan = new IntSpan("[7, 10)");
2     int value = 5 ;
3     intspan.is_adjacent(value); //false
```

Listing 3.34: Topological method for IntSpan

Position Operations

- The same operators as the ones presented during the last section were implemented. They take same parameter and perform a similar behavior tailored for `FloatSpan`/`IntSpan`. A code snippet describing one position operator is provided below:

```
1     IntSpan intspan = new IntSpan("[7, 10)");
2     int value = 5 ;
3     intspan.is_left(value); //false
4     intspan.is_over_or_right(value); //true
```

Listing 3.35: Position Operations for IntSpan

Set Operations

- Again, set operations are in all points the same as the last section set operators described.

3.3.3.3 FloatSpanSet and IntSpanSet

`FloatSpanSet` are a collection of non-overlapping `FloatSpan` objects, representing multiple floating-point ranges. Similarly, `IntSpanSet` are a non-overlapping set of `IntSpan` objects, which collectively represent a series of separate integer ranges.

Constructor

- Constructors for both classes are the same as the ones described for their parent abstract class `Span`. The unique difference between them is the fact that among the multiple constructors taking as parameter a number, `FloatSpan` will only accept float values, while `IntSpan` only integer values. Thus, it is not necessary to present them or detail them in this section.

Conversions

- Typically, a `FloatSpanSet`/`IntSpanSet` can be converted into a FloatSpan/IntSpan new object through the method to_span. Analogously, a FloatSpanSet can be converted to a new IntSpanSet object and vice-versa thanks to the to_intspanset, to_floatspanset methods.

Accessors

- The implementation of both classes gives an access to the first span of the set, the last span and width value respectively through the start_span, end_span and width methods.

Transformations and Topological operations

- Transformations and topological operators are exactly similar to the ones described in the past section, having the same behavior for the current types.

Position and Set operations

- The identical positions and set operators as the ones presented during the previous section were implemented. They take same parameter and perform a similar behavior tailored for `FloatSpanSet`/`IntSpanSet`. Redescribe them in this section, will be a redundant task.

3.3.4 TextSet

`TextSet` which is directly inherited form the `Set` type, as its name suggest, is concrete class that can store a set of `String`. The class structure is depicted on the Figure 3.9

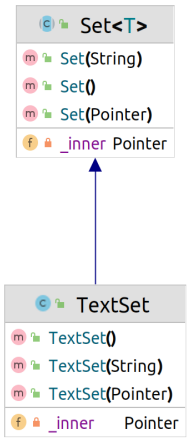

Figure 3.11: TextSet Class Structure

Constructor

1. The two classical constructors, a `String` and a `Pointer` constructors, were implemented allowing users to clearly create `TextSet`. It is noted that it is necessary to include opening and closing brackets to encompass the set of strings.

```
1    TextSet tset = new TextSet("{a, b, c, def}")
```

Listing 3.36: Constructor for TextSet

Accessors

- As for the `GeoSet` types, a star_element method, returning the first element of the set, and a end_element method, returning the last element of the set, were implemented.Additionally, a method named element_n and taking as parameter n (integer) allows the user to acquire the nth element of the set.

```
1    TextSet tset = new TextSet("{A, BB, ccc}");
2    tset.start_element(); // "A"
3    tset.element_n(2); // "B"
```

Listing 3.37: Accessors for TextSet

Transformations

- One specificity of `TextSet` type is the ability to transform it into lowercase string or uppercase string respectively with lowercase and uppercase method.

Set Operations

- Similarly to `GeoSet`, the union, intersection and subtraction between `TextSet` and `String` are made possible with the respective implemented methods.

3.3.5 Time

`Time` interface serves as a fundamental interface encapsulating the concept of temporal arrays/sets elements. This interface is implemented by `Period`, `PeriodSet` and `TimestampSet` types. The corresponding structure of the entire Time package is depicted in Figure 3.12.

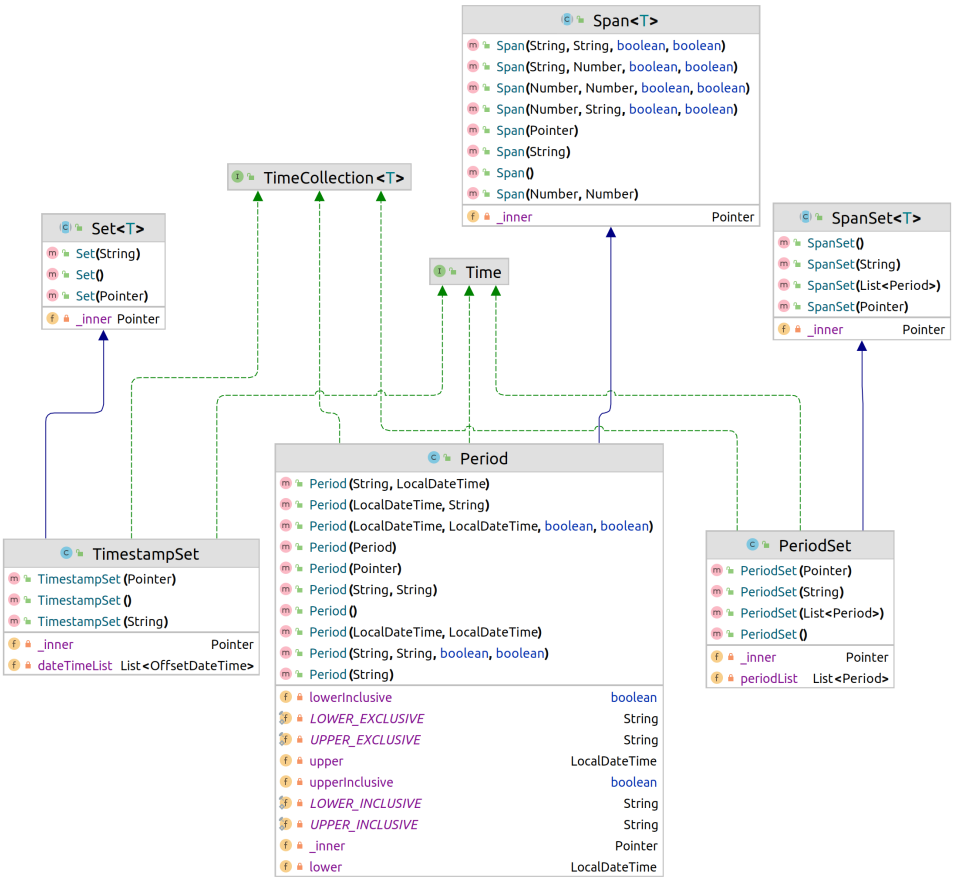

Figure 3.12: Time Package Structure

3.3.5.1 Period

A `Period` represents a time interval, defined by a start and an end time. Typically, such a class would be used to represent a continuous period during which a certain condition holds true or an event occurs. Being inherited from `Span` with `LocalDateTime` as `typename`, it encapsulates a start and end time, offering functionalities to query and manipulate this temporal `span`. The start and end time of a `Period` is represented by the `LocalDateTime` type from java in order to profit of its capabilities.

Constructor

The `Period` types implements 9 different constructors that are directly related to the parent class. By default, the lower bound is inclusive, while the upper bound is exclusive.

1. The first constructor and the easiest one is the `String` constructor. It allows a user to construct the `Period` object with a simple string:

```
1    Period period = new Period("(2019-09-08 00:00:00+00,
        2019-09-10 00:00:00+00)");
```

Listing 3.38: String Constructor for Period

2. The second interesting constructor takes as parameter the lower and the upper bound as two separate `String`. Informally, the object constructed will cast the two strings into `OffsetDateTime`:

```
1    Period period = new Period("2019-09-08 00:00:00+0",
        "2019-09-10 00:00:00+0");
```

Listing 3.39: Second String Constructor for Period

3. The third one is the `Pointer` constructor that will be useful while constructing a `Period` object from inner MEOS objects.

```
1    Period period = new Period(inner);
```

Listing 3.40: Pointer Constructor for Period

4. Concerning the last constructors, they are all a combination of `String` and `LocalDateTime` with bounds parameters. As for the second constructor, the `String` or `LocalDateTime` are casted into `OffsetDateTime` which is the only type accepted for `Period` object creation.

Accessors

- Multiple accessors are implemented in order to increase the manipulation of a `Period` object. For example, the lower and upper time is accessible through the methods lower and upper. The inclusivity of each bound can also be obtained through upper_inc and lower_inc functions.

```
1    Period period = new Period("(2019-09-08 00:00:00+00,
        2019-09-10 00:00:00+00)");
2    period.lower();
3    period.upper();
4    period.upper_inc();
```

Listing 3.41: Accessors for Period

Topological Operations

- Similarly to the `Span` type, it is possible to perform topological operations on `Period` objects. One can test for adjacency, capacity, identity or overlapping of a `Period` with regards to a `Temporal` object, a `Time` object or a `Box` object.

```
1     Period period = new Period("(2019-09-08 00:00:00+00,
         2019-09-10 00:00:00+00)");
2     STBox stbox = new STBox("STBOX XT(((1, 1),(2,
         2)),[2019-09-01,2019-09-02])");
3     PeriodSet pset = new PeriodSet("{[2019-09-01
         00:00:00+00, 2019-09-02 00:00:00+00], [2019-09-03
         00:00:00+00, 2019-09-04 00:00:00+00]}");
4     period.is_same(stbox);
5     period.overlaps(pset);
6     period.contains(pset);
```

Listing 3.42: Topological Operations for Period

Position Operations

- Four position methods are implemented for this type where a period can be positionally compared to another `Time` object. These operations would test if the first parameter was strictly before, after or if it is before, after allowing for overlaps. The names of the methods are respectively is_before, is_after, is_over_or_before, is_over_or_after.

```
1     Period period = new Period("(2019-09-08 00:00:00+00,
         2019-09-10 00:00:00+00)");
2     PeriodSet pset = new PeriodSet("{[2019-09-01
         00:00:00+00, 2019-09-02 00:00:00+00], [2019-09-03
         00:00:00+00, 2019-09-04 00:00:00+00]}");
3     period.is_after(pset);
4     period.is_before(pset);
```

Listing 3.43: Position Operations for Period

Set and Comparisons Operations

- `Set` operations are strictly similar to the set operations of its parent. The difference relies on the fact that it can take as parameter any `Time` object instance and returns a call to the corresponding `Span` method with the parameter.

3.3.5.2 PeriodSet

`PeriodSet` represents a collection of periods (as defined by the `Period` class), allowing for the representation of multiples, possibly non-contiguous, time intervals. The particularity is that the `PeriodSet` is of a fixed length, being non-empty with finite bounds. Since it is inherited from `SpanSet`, it offers functionalities to manipulate and operate multiple periods.

Constructor

The `PeriodSet` types implements 4 main different constructors that are directly related to `SpanSet`.

1. As for all concrete types that we are covered so far, the string constructor is implemented and requires thoroughness depending on the size of the set.

```
1    PeriodSet pset = new PeriodSet("{[2019-09-01 00:00:00+00,
        2019-09-02 00:00:00+00], [2019-09-03 00:00:00+00,
        2019-09-04 00:00:00+00]}");
```

Listing 3.44: String Constructor for PeriodSet

2. It is also possible to construct a `PeriodSet` through a list of `Periods`. This list must be a well formated Java List.

```
1    Period p1 = new Period("2019-09-08 00:00:00+0",
        "2019-09-10 00:00:00+0");
2    Period p2 = new Period("2019-09-08 00:00:00+0",
        "2023-09-10 00:00:00+0");
3    List<Period> list = new ArrayList<Period>();
4    list.add(p1);
5    list.add(p2);
6    PeriodSet pset = new PeriodSet(list);
```

Listing 3.45: List Constructor for PeriodSet

3. The third one is the `Pointer` constructor, being important for internal interactions with MEOS.

```
1    PeriodSet pset = new PeriodSet(inner);
```

Listing 3.46: Pointer Constructor for PeriodSet

4. It is also possible to construct a `PeriodSet` object form its WKB representation employing the following static constructor:

```
PeriodSet pset =
    PeriodSet.from_hexwkb("012200020000000300A01E4E7134
02000000F66B85340200030060CD899934020000C0A4A7AD340200");
```

Listing 3.47: From Hexwkb Constructor for PeriodSet

Accessors

- A `PeriodSet` containing multiple periods, one can access the first and the last `period` of the `periodset` through the methods start_period and end_period. The first and the last timestamp of the entire `PeriodSet` are also accesible through the start_timestamp and end_timestamp. Moreover, the number of timestamps as well as the hash representation is made available with appropriated methods.

```
PeriodSet pset = new PeriodSet("{[2019-09-01 00:00:00+00,
    2019-09-02 00:00:00+00], [2019-09-03 00:00:00+00,
    2019-09-04 00:00:00+00]}");
long value = pset.hash();
Period p = pset.start_period();
LocalDateTime ltime = pset.end_timestamp();
```

Listing 3.48: Accessors for PeriodSet

Topological, Positions, Set and Comparisons Operations

- Concerning these four groups of operations, it is not necessary to describe them in this section since the same operators and methodology of implementation as the `Period` type is put in place. The only difference is that it is specific to a `PeriodSet` type, using others MEOS functions.

3.3.5.3 TimestampSet

Inherited from `Set<LocalDateTime>`, this class represents a collection of discrete points in time. Each element in the set is a distinct timestamp (`LocalDateTime`), and the class provide methods to handle this collection of timestamps efficiently thanks to its inheritance. This type contains one dimension and enforces that no duplicates can be found within the set.

Constructor

The `TimestampSet` types implements also 4 main constructors that are directly related to `Set`.

1. The `String` constructor, which is the advised constructor for quick object instantiation, takes a string representing a set of timestamp as parameter.

```
TimestampSet tset = new TimestampSet("{2019-09-01
    00:00:00+0, 2019-09-02 00:00:00+0, 2019-09-03
    00:00:00+0}");
```

Listing 3.49: String Constructor for TimestampSet

2. The construction of a `TimestampSet` through a `List` of `LocalDateTime`. This list must be a well formated Java List.

```
LocalDateTime l1 = LocalDateTime.now();
LocalDateTime l2 = LocalDateTime.now();
List<LocalDateTime> list = new ArrayList<LocalDateTime>();
list.add(l1);
list.add(l2);
TimestampSet tset = new TimestampSet(list);
```

Listing 3.50: List Constructor for TimestampSet

3. The third one is the `Pointer` constructor, being important for internal interactions with MEOS.

```
TimestampSet tset = new TimestampSet(inner);
```

Listing 3.51: Pointer Constructor for TimestampSet

Conversions

- Two conversions methods exist for the current type. A first one that converts a `TimestampSet` to a `PeriodSet` thanks to the to_periodset method. The second one that encompasses it to a `Period` with to_span:

```
TimestampSet tset = new TimestampSet("{2019-09-01
    00:00:00+0, 2019-09-02 00:00:00+0, 2019-09-03
    00:00:00+0}");
PeriodSet pset = tset.to_periodset();
Period p = tset.to_period();
```

Listing 3.52: Conversions for TimestampSet

Accessors

- A `TimestampSet` containing multiple timestamps, one can access the first and the last `timestamp` of the timestampset through the methods start_element and end_element. In addition, the number of timestamps as well as the hash representation is made available with appropriated methods.

```
1    TimestampSet tset = new TimestampSet("{2019-09-01
        00:00:00+0, 2019-09-02 00:00:00+0, 2019-09-03
        00:00:00+0}");
2    long value = tset.hash();
3    LocalDateTime p = pset.start_element();
```

Listing 3.53: Accessors for TimestampSet

Topological, Positions, Set and Comparisons Operations

- Concerning these four groups of operations, it is not necessary to describe them in this section for the aforementioned reasons as the previous section that described the `PeriodSet` type. The unique difference is that it is specific to a `PeriodSet` type, using others MEOS functions.

3.4 Temporal Types

This section will treat of all temporal related types, in the sense of MobilityDB, that can be found under the `temporal` package and the `basic` package. The location of this latter is depicted in Figure 3.13.

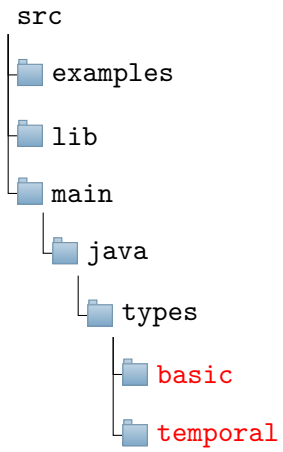

Figure 3.13: Temporal and Basic Packages Location

MobilityDB, as an extension of PostgreSQL and PostGIS, introduces a significant advancement in the management of temporal data. It provides a range of temporal types designed to capture the evolution of various base types over time. This feature is vital in representing dynamic data, such as the changing temperature in a room or the fluctuating number of people on a train. In MobilityDB, these dynamic values are encapsulated within temporal types such as `tfloat`, `tint`, `tgeompoint` or `tgeogpoint`.

Temporal types, including `tbool`, `ttext`, `tint`, `tfloat`, `tgeompoint`, and `tgeogpoint`, are fundamentally based on standard types like bool, text, int, float, and the PostGIS types for 2D and 3D points. This diversity of types allows for a extensive range of temporal data representations, from simple boolean values to complex geographic points. The essence of these temporal types lies in their ability to describe the temporal dimension of data, offering a nuanced view of data evolution.

3.4.1 Temporal

In the context of JMEOS, the Temporal type is an abstract class that lays the foundational structure for handling temporal data. It implements the `TemporalObject` interface, which is also shared by other types dealing with temporal dimensions. This design decision facilitates method parameters to accept a "union" of multiple types, which is particularly useful given Java's strict typing system. The `Temporal` class in JMEOS encompasses a variety of methods categorized into groups like accessors, transformation, modificaitons, restrictions, topological, position, similarity and comparisons operations, providing a comprehensive toolkit for temporal data manipulation. The Figure 3.14 shows the overall structure of the abstract temporal types.

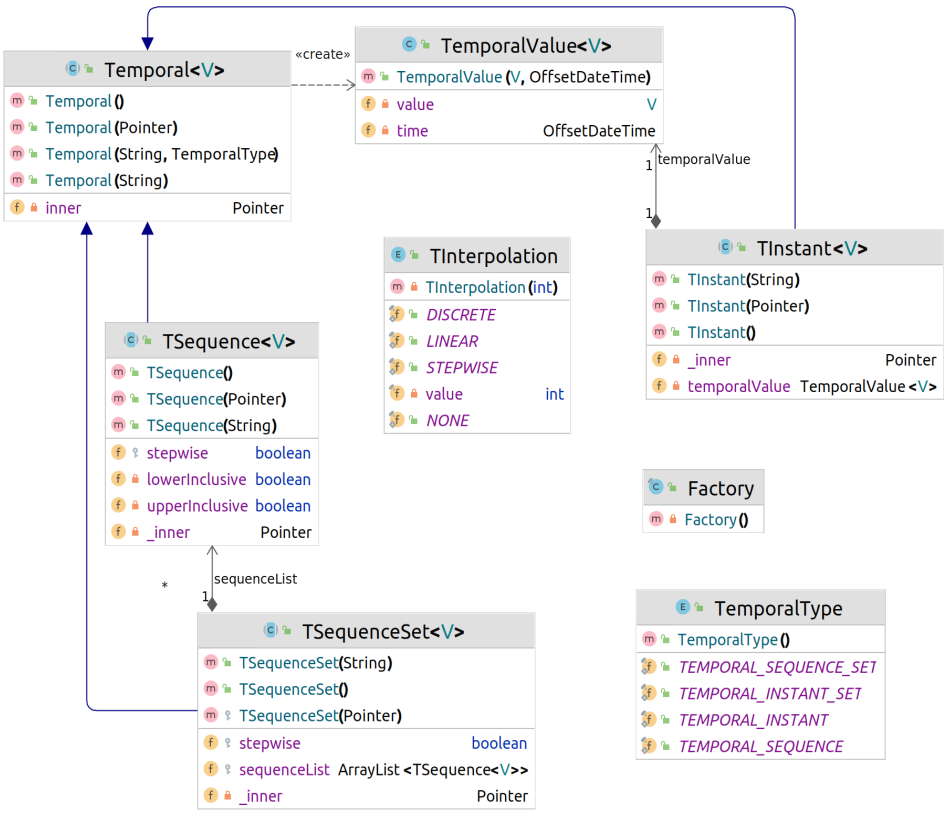

Figure 3.14: Class Structure of Temporal Abstract classes

Building upon the `Temporal` base class, JMEOS introduces three main abstract subclasses: `TSequence`, `TInstant`, and `TSequenceSet`. `TSequence` represents a temporal sequence, capturing a series of data points over a `period`. `TInstant`, on the other hand, denotes a particular temporal instant, a snapshot of data at a specific moment. `TSequenceSet` extends the concept to a set of temporal sequences, allowing for the representation of disjointed or intermittent data sequences. These subclasses inherit from the `Temporal` class, thus inheriting its elaborate set of functionalities.

The final step in the hierarchy involves the creation of main types: `tbool`, `tint`, `tfloat`, `ttext`, `tgeog`, and `tgeom`. These types are tailored to specific data forms, from boolean values and text to geographic and geometric points. Each of these main types is developed on the structure provided by the `TSequence`, `TInstant`, and `TSequenceSet` classes, ensuring consistency and robustness in temporal data handling.

3.4.2 TInstant

A `TInstant` in JMEOS is a concise way to represent a specific data point, of a certain type, at a precise moment. It follows the nomenclature:

$$v@t$$

, where v is a boolean, integer, float, text, geometric or geography type and t is the timestamp. For instance, to document the occupancy of a train car at a specific moment, say 50 people at 10:30 AM on July 4th, 2020; it would be represented as "50@2020-07-04 10:30:00". This format of `TInstant` allows for the precise and unambiguous recording of data at an exact timestamp, which is essential for temporal data analysis in JMEOS.

3.4.3 TSequence

`TSequence` in JMEOS offer a dynamic view of how values evolve over time, enabling interpolation of values at any point within the sequence. This feature distinguishes them from temporal instant sets, as they provide a continuous representation of value changes over a specified duration. The method of interpolation is based on the underlying data type: discrete for types such as bool, text, and integer, and linear for float, geometry, and geography.

Typically, one can consider a scenario where tracking of occupancy of a conference room throughout a day using a tint, is put in place. This temporal sequence logs the number of people entering or leaving the room, recording a new temporal instant each time there is a change in occupancy. For instance, if 10 people are in the room at 9:00 AM, 15 at 9:30 AM after a meeting starts, and then 5 at 10:00 AM when the meeting ends, this sequence might be represented as:

$$[10@9:00, 15@9:30, 5@10:00]$$

Here, the number of people in the room at any given time is determined by the most recent count before that time, employing stepwise interpolation. This method is well-suited for scenarios where values change discretely at specific moments, as is the case with room occupancy, where people enter or exit at distinct times.

Another scenario is monitoring the water level of a reservoir. Regular measurements of the water level (`tfloat`) allow for a linear interpolation between readings. This continuous interpolation method provides a nuanced picture of gradual fluctuations in the water level.

A temporal sequence with four data points might be depicted as:

$$[v0@t0, v1@t1, v2@t2, v3@t3]$$

The sequence's boundaries, denoted by '[' and ']' for inclusivity or '(' and ')' for exclusivity, specify whether the endpoints are part of the sequence. This structure is offering an interpolated and continuous view of changes over time.

3.4.4 TSequenceSet

A temporal value with sequence set duration in JMEOS captures the evolution of values across multiple sequences, with undefined values between these sequences. This structure is particularly useful for representing discontinuous data over time.

For example, a `TSequenceSet` could represent temperature readings taken at different periods of a day but not continuously throughout the day. In this case, two sequences with two instants each might be defined. The first sequence could represent morning temperatures, and the second sequence could represent evening temperatures. This would be denoted as:

$$[v0@t0, v1@t1], [v2@t2, v3@t3]$$

The square brackets indicate the inclusiveness or exclusiveness of the bounds in each sequence.

3.4.5 General Structure: Factory, Interpolation and TemporalType

In Java, multiple inheritances are not allowed, primarily to avoid the diamond problem, which arises from the ambiguity in shared attributes and methods originating from a common base class. This limitation shapes how we design class hierarchies, particularly for representing temporal types.

Given this constraint, the approach adopted for JMEOS involves using inheritance and interfaces judiciously. For the temporal types discussed in the subsequent sections, each class inherits from one of the three primary temporal classes: `TInstant`, `TSequence`, or `TSequenceSet`. These classes, in turn, inherit from the abstract Temporal class. The base types, such as `TGeomPoint` and `TGeogPoint`, are represented as interfaces (akin to abstract classes in Java) rather than concrete classes, since they encapsulate behaviors common to all temporal types but do not themselves implement specific behaviors.

To facilitate the creation of temporal objects, a special class named `Factory` is implemented. Illustrated in Figure 3.14, this class sole purpose is to instantiate temporal objects based on provided parameters. It is particularly valuable in the context of abstract classes or interfaces in Java, where direct instantiation of concrete classes is not possible. The `Factory` class serves as a crucial point for creating new instances of temporal objects with their specific temporal types and base class configurations.

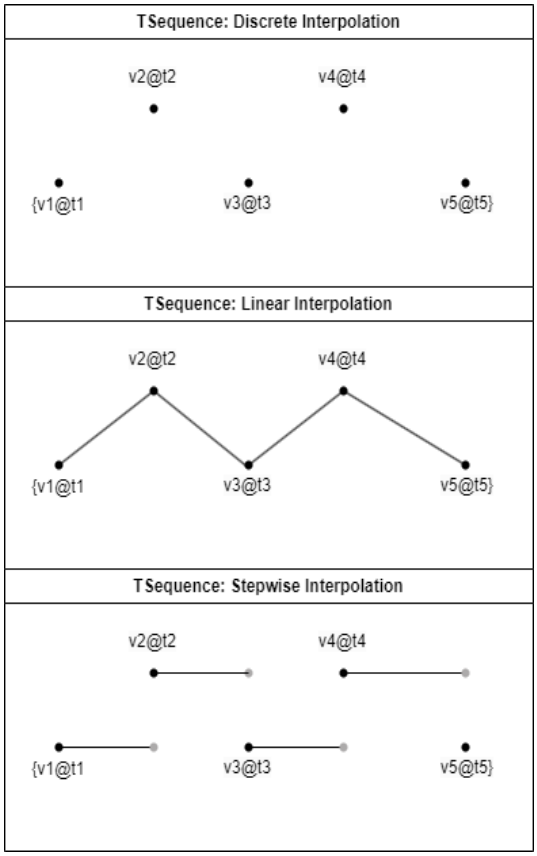

Figure 3.15: Interpolation types for TSequence

Another component in the JMEOS design is the `Interpolation` class. This class specifies the type of interpolation applied to temporal sequence instances. For instance, `TSequence` subtypes can have discrete, linear, or stepwise interpolation, as shown in Figure 3.15. While `TSequenceSet` can only have linear or stepwise interpolation.

`TemporalType` is an enumeration that stores the subtype of a `Temporal` type. This enumeration is useful to quickly and clearly creates new `Temporal` type through the `Factory` method.

3.4.6 TBool

A `TBool` represents a temporal boolean value, encapsulating a boolean state that changes over time. It's typically represented as *boolean*@*t*1, where boolean is a boolean value (true or false), and t1 is a timestamp marking the point in time when this value is valid.

For instance, consider recording whether a light is on or off at different times. A `TBool` representation could be **true@2020-01-01 08:00:00+00**, indicating that the light was on at 8 AM on January 1, 2020. In the JMEOS library, `TBool` is implemented through three concrete classes: `TBoolInst`, `TBoolSeq`, and `TBoolSeqSet`. Each of these

classes is derived from the corresponding temporal subtype. While these concrete classes handle the instantiation and specific details like constructors for strings and pointers, the main functionalities and behaviors of these types are defined in the `TBool` interface. This interface lays out the practical operations and methods applicable to temporal Booleans, irrespective of their specific temporal subtype (instant, sequence, or sequence set). The overall structure is detailed in Figure 3.16.

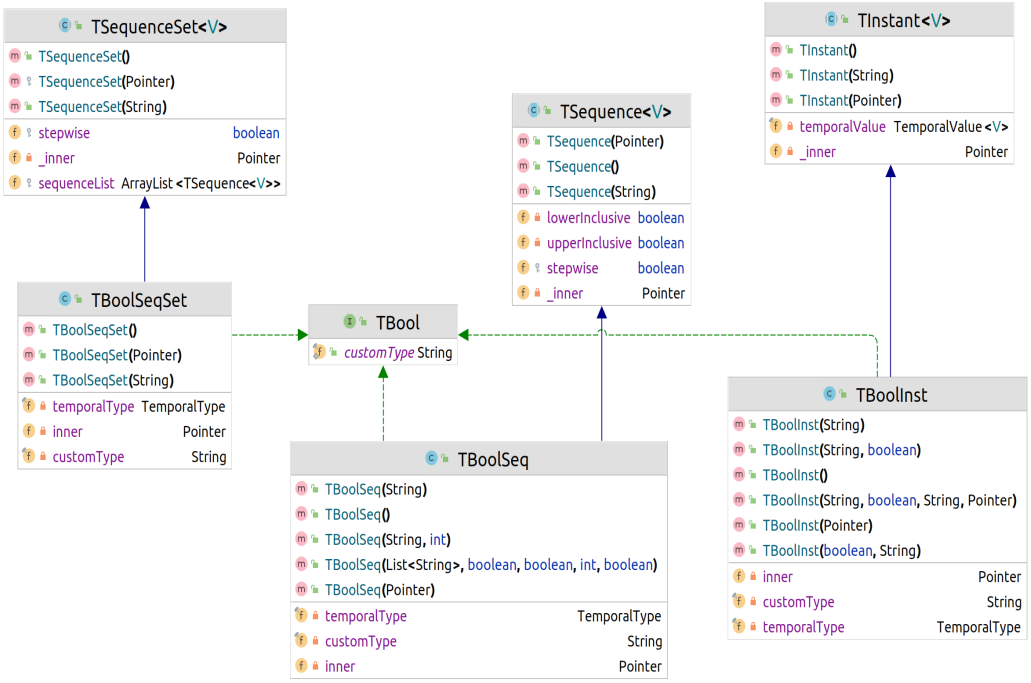

Figure 3.16: TBool overall structure

Constructors

- Apart from the `String` and `Pointer` constructors available in the concrete classes, one can create a new `TBool` object either from a Time instance or another `Temporal` instance respectively with the from_base_time and from_base_temporal methods.

```
1    Period base = new Period("[2019-09-01, 2019-09-02]");
2    TFloatInst tinst = new TFloatInst("1@2019-09-01");
3    TBoolInst tb = (TBoolInst) TBool.from_base_time(true,
        base);
4    TBoolInst tb2 = (TBoolInst)
        TBool.from_base_temporal(true, tinst);
```

Listing 3.54: Constructors for TBool

Accessors

- Since a `TBool` encompasses `TBoolSeq` and `TBoolSeqSet`, it could be useful to obtain the first and the last value of a sequence or sequence set in order to clearly identify the bounds. In JMEOS, these operations are possible through the start_value and end_value.

```
1    TBoolSeq tseq = new TBoolSeq("[True@2019-09-01,
        False@2019-09-02]");
2    tseq.start_value(); //true
3    tseq.end_value(); //false
```

Listing 3.55: Accessors for TBool

Ever and Always Comparisons Operations

- Ever and Always comparisons are another type of operations that are interesting to perform. It allows anyone to check if an `TBool` object is always, ever or never equal to the boolean predicate.

```
1    TBoolSeq tseq = new TBoolSeq("[True@2019-09-01,
        False@2019-09-02]");
2    tseq.always_eq(true); //false
3    tseq.ever_eq(false); //true
4    tseq.never_eq(true); //false
```

Listing 3.56: Ever and Always for TBool

Temporal and Restrictions Operations

- Temporal comparisons allow to compare a `TBool` object with a boolean predicate. Through the methods temporal_equal_bool and temporal_not_equal_bool, one can assess the equality and inequality of a `TBool` with regards to a boolean.

  Restrictions operators are similar to set operations in the sense that one can restrict an object to a boolean predicate with at_bool or restrict it to the complement of this predicate, minus_bool.

Boolean Operations

- Concerning the boolean operations group of methods, one can perform a temporal conjunction with another `TBool` object or a boolean value thanks to the temporal_and, temporal_and_bool methods. Similarly, a temporal disjunction can be obtained with temporal_or, temporal_or_bool methods. Moreover, the methods when_true and when_false allows a user to obtain a `PeriodSet` regrouping all timestamps that matches with the predicate.

3.4.7 TInt and TFloat

In order to define common functionalities and behaviour for both `TInt` and `TFloat` types, they extend an interface called `TNumber`. The methods implemented in this latter will be described in the methods groups sections below.

`TInt` and `TFloat` represent the evolution of integer and float values over time. These types are particularly useful for tracking changes in numerical data across different time points.

`TInt` encapsulates integer values that vary over time. Typically, it is represented as *integer@t*, where integer is an integer value and t is a timestamp. For example, **5@2020-01-01 10:00:00+00** indicates that the value was 5 at 10 AM on January 1, 2020. This can be useful for scenarios like tracking the number of people in a room at different times.

`TFloat` functions similarly but for floating-point numbers, represented as *float@t*, where float is a floating-point value and t is a timestamp. For instance, **23.5@2020-01-01 10:00:00+00** indicates a value of 23.5 at the specified time. `TFloat` types can be useful in contexts like environmental data monitoring, where values such as temperature or humidity are recorded at different times.

Like `TBool`, the main functionalities and behaviors for these types are outlined in their respective interfaces,i.e. `TFloat` and `TInt`, with concrete classes focusing on specific instantiation details. The overall structure is depicted on Figure 3.17.

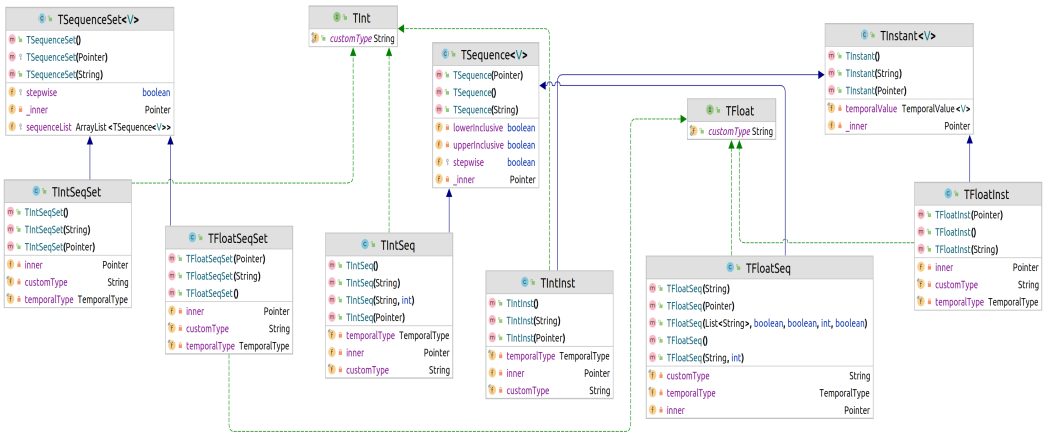

Figure 3.17: TFloat and TInt overall structure

This section presents both types at the same time because they share the same functions but adapted to their base types.

Constructors

- Analogously to `TBool` types, it is also possible to construct a `TFloat` or `TInt` types from a temporal or time object thanks to the same method previously discussed.

Conversions Operations

- When possible, conversions between `TInt` and `TFloat` might be useful to perform specific operations. Thus, the to_tint and to_tfloat methods allows such operations in JMEOS.

```
TFloatInst tf = new TFloatInst("1,5@2019-09-01");
TIntInst ti = new TIntInst("1@2019-09-01");
tf.to_int();
ti.to_tfloat();
```

Listing 3.57: Conversions Operations for TFloat and TInt

Accessors

- These two types being numbers, one can be interested to access the span/spans value of the object, as described in this Chapter, thanks to the methods value_span, value_span. One can also be willing to obtain the first and the last value, similar to a `TBool`, but also the minimum and maximum one with min_value and max_value. In addition to that, it is possible to obtain the time weighted average of the object through time_weighted_average.

```
TFloatInst tf = new TFloatInst("1,5@2019-09-01");
tf.min_value(); //1.5
tf.max_value(); //1.5
```

Listing 3.58: Accessors for TFloat and TInt

Ever and Always Comparisons Operations

- Ever and Always comparisons are another type of operations that are interesting to perform. It allows anyone to check if a `TFloat` or `TInt` object is equal, always, ever, never, less, greater than a int/float value.

```
TFloatInst tf = new TFloatInst("1,5@2019-09-01");
tf.always_eq(1.0); //false
tf.ever_eq(0.0); //false
tf.never_eq(10.0); //false
tf.ever_greater_or_equal(1.0); //true
```

Listing 3.59: Ever and Always for TFloat and TInt

Restrictions

- Closely related to restrictions methods of `TBool`, one can strictly restrict or restrict to the complement, an object, with regards to a value. This value can be any object defined in the Number package of the Collections package, i.e. a `FloatSpan`, `IntSpan`, `FloatSet`, `IntSet`, `FloatSpanSet`, `IntSpanSet`.

Positions Operations

- The positions operations performed on a `TFloat/TInt` are taken back from the TBox type. A conversion is made with the methods bounding_box before the effective application of the positional method.

Mathematical Operations

- Mathematical operations concern the basic operations that could be done between two integer/float value. For example, one can add, divide, subtract or multiply an Integer/Float to a `TInt/TFloat`.

Temporal and Transformations Operations

- Temporal comparisons allow to compare a `TFloat` or `TInt` object with a value. Through the methods temporal_equal_number and temporal_not_equal_number, one can assess the equality and inequality of a `TFloat/TInt` with regards to a float/int value.

  Transformations operators are only present for `TFloat`. It allows to convert a TFloat object an other one as degrees or radians.

3.4.8 TText

`TText` is a temporal type that represents the evolution of string values over time. `TText` is formulated as $text@t$, where text is a string value and t is a timestamp. An example of a `TText` representation could be **"Open"@2020-01-01 09:00:00+00**, indicating that a store was "Open" at 9 AM on January 1, 2020. Again like TBool, all functionalities and behaviors for the subsequent `TText` types are defined in the `TText` interface, as described in Figure 3.18.

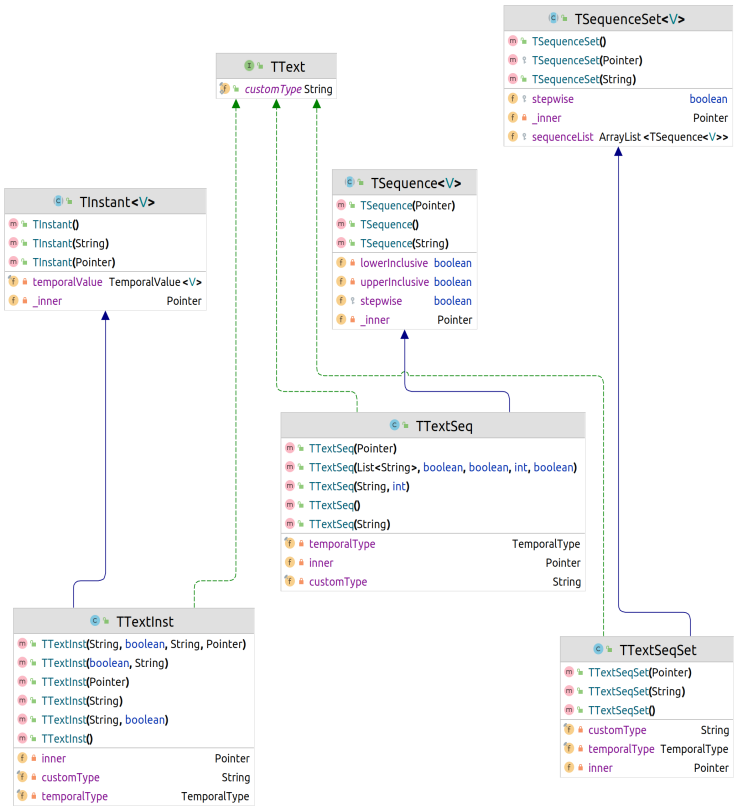

Figure 3.18: TText overall structure

Constructors, Accessors, Ever and Always and Temporal Operations

- For `TText`, it was decided to not described the implementation that was done simply because the methods of the different groups are already defined and described in the past sections. The only difference that is contained in `TText` regarding these methods is the fact that it is specific to the `String` base type.

3.4.9 TPoint

A `TPoint` represents a temporal point data type. Typically used to describe the movement or change of geographical points over time, it is expressed following this formula: $p@t$, where p is a Point generated from the jts library and t is a timestamp.

The `TPoint` interface will serve as a base interface for the `TGeomPoint` and `TGeogPoint` interfaces from which the concrete classes, derived from the temporal subtype, are implemented. Three abstract classes deriving from the three temporal subtype are implementing the `TPoint`. This choice was made in order to allow the concrete classes to indirectly derive from the main Temporal abstract class and so profit of it's functionnalities. The overall structure is depicted on the Figure 3.19

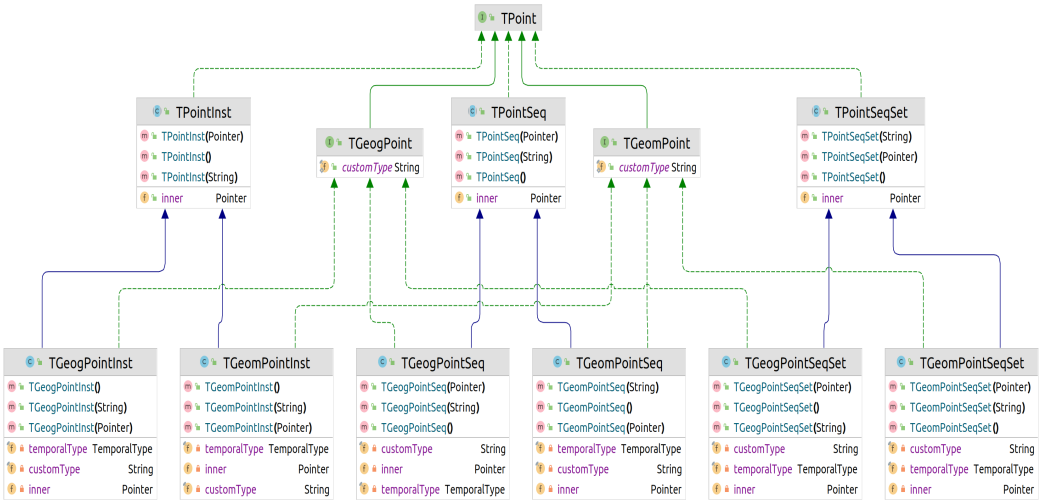

Figure 3.19: TPoint overall structure

Output Operations

- The group of operation are the output operations allowing a user to visually observe the value of a `TPoint`, and so all of the implementing classes. Typically, one can output a `TPoint` as a string or WKT with to_string or as_ewkt. It can also be outputed as an EWKT which introduces the **"SRID;"** string at the beginning of the `TPoint` with as_ewkt, or on a GeoJSON format with as_geojson.

```
1    TGeomPointInst tgeom = new TGeomPointInst("Point(1.5
         1.5)@2019-09-01");
2    tgeom.to_string();
3    tgeom.as_ewkt();
4    tgeom.as_geojson();
```

Listing 3.60: Output Operations TPoint

Accesors Operations

- Concerning the accessors operations, the first and start value are accessible as for the other temporal objects inheriting from `Temporal`. In addition, since points are used to represent moving objects, the speed, the length, the tridimensional position (x,y,z), the angular difference, the azimuth and the direction are among others reachable respectively through the methods speed, length, x, y, z, angular_difference, azimuth and direction.

```
1    TGeomPointInst tgeom = new TGeomPointInst("Point(1.5
        1.5)@2019-09-01");
2    tgeom.speed();
3    tgeom.azimuth();
4    tgeom.angular_difference();
5    tgeom.x();
```

Listing 3.61: Accessors Operations TPoint

Spatial Reference System, Restrictions, Positions Operations

- Working with geographical and geometrical points involves that anybody can access the spatial reference identifier (SRID) with srid, but also modify it with set_srid methods. Concerning the restrictions operators, they are the same as the `TBool` type. The positions operations are still derived from the STBox after conversion.

Ever Spatial and Temporal Spatial Relationships Operations

- Ever spatial relationships operations are similar to ever and always operations described for `TBool`. It compares two `TPoint` for capacity, intersection or disjoint operations and returns a boolean value depending on the function. Typically, it takes as parameter a Geometry object (jts library), an `STBox` or an other `TPoint`. Concerning the temporal spatial relationships, it tests disjunction, intersections or crossing between a `TPoint` and a `STBox`/Geometry object. It returns a `TBool` object containing the answer of the predicate alongside with the timestamp associated to the predicate.

3.4.9.1 TGeogPoint and TGeomPoint

As previously mentioned, `TGeogPoint` and `TGeomPoint` are two interfaces that implements the `TPoint` interface. These two interfaces describe a specific behavior for geographic and geometrical points respectively.

Constructors, Ever, Always and Temporal Comparisons Operations

- All of the methods implemented for each group are strictly similar to the ones that were already presented except that it is specific for `TGeogPoint` or `TGeomPoint`.

3.5 Dockerization

As discussed in Chapter 2 of the thesis, the dockerization of JMEOS represents a significant step in enhancing the usability and deployment efficiency of the project. The JMEOS Docker container implementation is hosted on GitHub: https://github.com/nmareghn/Docker-JMEOS/tree/main . This repository provides the Dockerfile and related scripts necessary to build and run the JMEOS environment in an isolated container.

Before proceeding with JMEOS containerization, users must install Docker and Git on their systems. Docker's platform-independent nature allows for seamless setup regardless of the underlying operating system. This installation step is crucial as it lays the foundation for building and running the JMEOS container.

To deploy JMEOS in a Docker environment, the following steps are followed:

- Clone the Docker files from the specified repository:

```
1 john@doe:~$ git clone
    ↪ https://gitlab.com/asded/docker_mobilitydb-jmeos
2 john@doe:~$ cd docker_mobilitydb-jmeos/.devcontainer
```

Listing 3.62: Cloning Docker Image

- Navigate to the directory containing the Dockerfile.
- Build the Docker container using:

```
1 john@doe:~/Docker-JMEOS/.devcontainer$ docker build -t
    ↪ mbjmeos:lasted .
```

Listing 3.63: Building Docker

- Run the container using Docker's run command:

```
1 john@doe:~/Docker-JMEOS/.devcontainer$ docker run -ti
    ↪ mbjmeos:lasted
```

Listing 3.64: Running Docker

The Dockerfile for JMEOS, based on the lightweight Linux distribution image **debian:bookworm-slim**, summarizes the process of setting up the environment for the library, which includes:

1. Installing necessary system dependencies like Git, CMake, PostgreSQL server development files, and PostGIS.
2. Setting up Amazon Corretto, a Java Development Kit (JDK) distribution, which ensures a stable and optimized Java runtime for JMEOS.

3. Building MobilityDB with MEOS, from its source to ensure integration with JMEOS.
4. Incorporating Maven for project management, crucial for Java-based projects like JMEOS.
5. Adding the JMEOS project to the container, including copying **libmeos.so** file, and cleaning up to reduce the container size.

3.6 Javadoc

Javadoc is an essential tool for Java programmers, providing a standardized way to document Java code. It generates HTML documentation from Java source code with special documentation comments. The significance of Javadoc in JMEOS cannot be overstated, as it greatly aids in understanding, maintaining, and scaling the software.

At its core, Javadoc comments facilitate clarity and consistency in code documentation. Javadoc is designed to describe the API of the classes, methods, constructors, and fields to those who want to understand how to use them [16]. Furthermore, Joshua Bloch in "Effective Java" emphasizes the importance of good documentation practices, including the use of Javadoc for maintaining high-quality code [8].

JMEOS being a complex project, Javadoc is particularly important. It ensures that all functionalities and intricacies of the code are well-documented, making the codebase accessible and comprehensible to new developers and contributors. The use of Javadoc in JMEOS includes several tags, each serving a specific purpose:

- **@param** provides descriptions for method parameters.
- **@author** attributes the creation of a class or method to a specific developer.
- **@since** indicates the software version that introduced a particular element.
- **@return** describes the return type of methods.
- **@throws** or @exception documents the exceptions a method might throw.
- **@link** creates hyperlinks to other elements in the documentation for quick reference.

In JMEOS, generation Javadoc is handled with Maven, using the command:

```
1 john@doe:~/JMEOS$ mvn javadoc:javadoc
```

Listing 3.65: Javadoc Generation

This command conventionally places the documentation in the target directory. However, in JMEOS, a decision was made to relocate this documentation to the `/docs` directory. This strategic move not only organizes the project structure but also makes the documentation easily accessible for users.

The Javadoc can be viewed by opening the **allpackages-index.html** file in a browser, as shown in Figure 3.20 . This file is the gateway to the entire documentation, presenting a comprehensive overview of all the classes, interfaces, methods and fields within JMEOS.

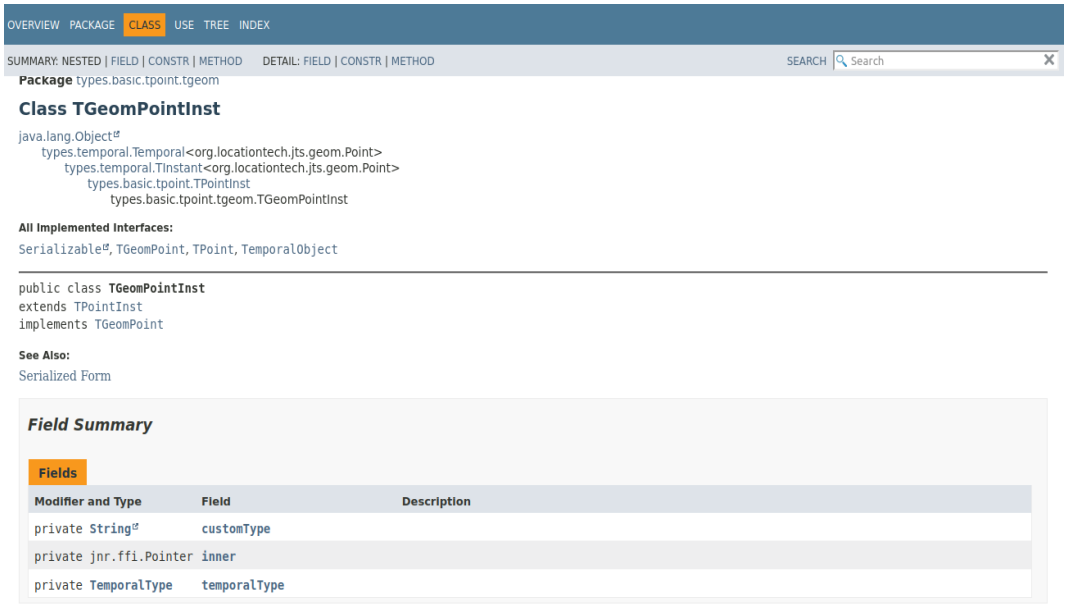

Figure 3.20: Java Documentation of the `TGeomPointInst` Class

Chapter 4

Unit Tests

4.1 Movitations and Structure

In the chapter dedicated to Unit Tests, it is substantial to clarify that the focus is not on providing detailed explanations for each individual test file. Instead, the chapter aims to offer a broader overview of the entire testing approach. This perspective encourages readers grasp the comprehensive nature of the testing process in the context of the project.

Specifically, the project encompasses 16 distinct test files, collectively running more than 1,000 unit tests. This extensive testing framework underscores the thoroughness and rigor applied to ensure the reliability and robustness of the project's components.

It is essential to acknowledge that not all files in the project have corresponding test files. This absence of direct unit tests for certain files is attributed to a few key reasons:

- **Irrelevance for specific types**: Some types or components within the project do not necessitate individual unit tests. This could be justified to their simplistic nature, where testing would not significantly contribute to the assurance of their functionality or reliability.
- **Non-testable entities**: There are elements within the project, such as abstract classes and interfaces, that are not directly testable. These constructs frequently provide foundational structures or generalized behaviors, and their effectiveness is typically validated indirectly through the testing of concrete classes that extend or implement them.
- **Implicit integration in other classes**: Certain functionalities, like methods of the Temporal abstract classes, are inherently integrated into other classes. As a result, their testing occurs indirectly when these other classes are tested. This approach ensures that the functionalities are validated in the context of their actual use within the system, providing a more realistic assessment of their performance and behavior.

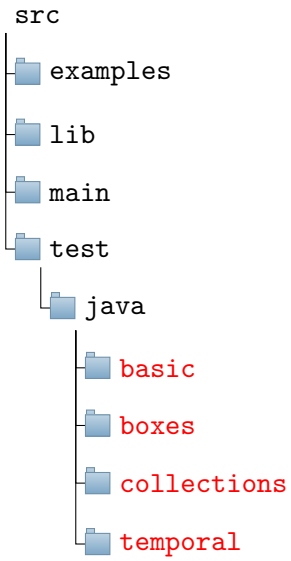

Figure 4.1: Test Package Structure

Unit testing carry out a crucial role in the development of JMEOS for several reasons, and its importance is supported by various academic sources:

- Unit tests in JMEOS help ensure that individual components function correctly. They validate that each module performs as expected under various conditions. This approach to testing is vital for maintaining high-quality code, particularly in a library like JMEOS, where reliability is paramount [24].
- With unit tests, developers can confidently initiate changes or refactor code, knowing that tests will immediately catch any errors introduced by changes. This aspect is especially critical in JMEOS, which interfaces with the complex MEOS library [14].
- Writing unit tests encourages developers to consider the structure and design of the code, leading to more modular and maintainable codebases. In the context of JMEOS, where the code interacts with separate components of the MobilityDB ecosystem, this is essential for long-term project sustainability [15].

4.2 JUnit

JUnit is chosen due to several reasons:

1. JUnit is one of the most popular testing frameworks in the Java ecosystem, which means it has a large community support and numerous assets for troubleshootings.

2. It integrates seamlessly with most Java development environments and build tools, including Maven, used in this project. This integration simplifies the testing process, making it more efficient and less prone to errors.

3. JUnit provides a comprehensive suite of testing tools and annotations that cater to a extensive range of testing needs, from basic assertions to more complex testing scenarios. This flexibility is particularly beneficial for a library like JMEOS, which might require varied testing approaches for various components.

4. It works well with CI/CD pipelines, facilitating automated testing alongside development. This compatibility is crucial for maintaining the ongoing quality and stability of JMEOS.

4.3 Executions

Executing unit tests in JMEOS is straightforward, notably when using Maven as the build tool. It simplifies the testing process, allowing developers to run all tests or target specific tests with ease. To run all unit tests, one can execute the following command:

```
john@doe:~/JMEOS$ mvn test
```

Listing 4.1: Maven Overall Testing

To test a specific file encompassing multiple tests:

```
john@doe:~/JMEOS$ mvn test -Dtest="FileTest"
```

Listing 4.2: Maven File Testing

To test a specific method in a class:

```
john@doe:~/JMEOS$ mvn test -Dtest="FileTest#method"
```

Listing 4.3: Maven Method Testing

Chapter 5

Code Analysis

5.1 Motivation

Static code analysis plays an essential role in software development, scrutinizing the source code without running it. This method helps developers spot potential problems like syntax errors, complexity, and security risks early in the development process.

A key benefit of static code analysis is its early problem detection capability, greatly reducing bug-fixing costs and efforts later in the development cycle [25]. Identifying issues before the code runs enhances the overall quality and dependability of the code, resulting in more robust, secure software [20].

Static code analysis is highly recommended as it augments dynamic testing approaches. While dynamic testing involves code execution to identify runtime errors, static code analysis offers a unique perspective by focusing on code structure and syntax, which might not lead to noticeable bugs during execution [11].

Additionally, static code analysis plays a vital role in improving code security. It spots typical security flaws like buffer overflows, SQL injections, and cross-site scripting, often missed in manual code reviews [38]. This aspect is crucial when developing applications that manage sensitive data or are integral to critical systems.

Moreover, static code analysis enhances code maintainability and readability. It promotes adherence to coding standards and best practices, leading to more uniform and comprehensible codebases. This consistency is crucial for long-term project viability, simplifying code changes and updates by various team members over time [33].

Integrating static code analysis tools, such as SonarQube, into the development workflow automates the review process. It offers continuous feedback and insights into code quality and can be incorporated into continuous integration pipelines. This integration ensures regular code quality checks, fostering a culture of quality and accountability among developers.

In conclusion, static code analysis is an essential component of contemporary software development. Its proactive stance in issue detection, adherence to coding norms, security enhancement, and maintainability improvement renders it a critical tool for creating high-quality, secure, and maintainable software.

5.2 SonarQube

For this project, SonarQube was chosen as the static code analysis tool, mainly because of its strong compatibility and effectiveness with Java-based projects. SonarQube stands out in detecting bugs, vulnerabilities, and code smells in Java as a comprehensive code quality management platform, making it an ideal match for this development setting [2].

A significant strength of SonarQube is its detailed rule set designed for Java. It is capable of identifying a wide array of issues, ranging from basic syntax errors to complex design issues and even security vulnerabilities specific to Java applications [27]. This capability positions SonarQube as a crucial asset in upholding high standards of code quality and security in Java projects.

Furthermore, SonarQube's integration features are noteworthy. It smoothly fits into various development tools and CI/CD pipelines, allowing for ongoing code quality assessment. This integration promotes an active stance in code quality management, helping to address potential problems quickly during development [6].

Nonetheless, SonarQube has its challenges. A potential downside is its demand for substantial resources. Conducting thorough scans on extensive codebases can be lengthy and might necessitate considerable computational power. This aspect can be problematic in continuous integration settings where fast build times are essential.

Another drawback is the learning curve for new users. Despite offering detailed insights and a user-friendly dashboard, it may take time for users to become accustomed to its interface and functionalities. The complexity in setting up custom rules and grasping the depth of its analysis could also be daunting for some developers.

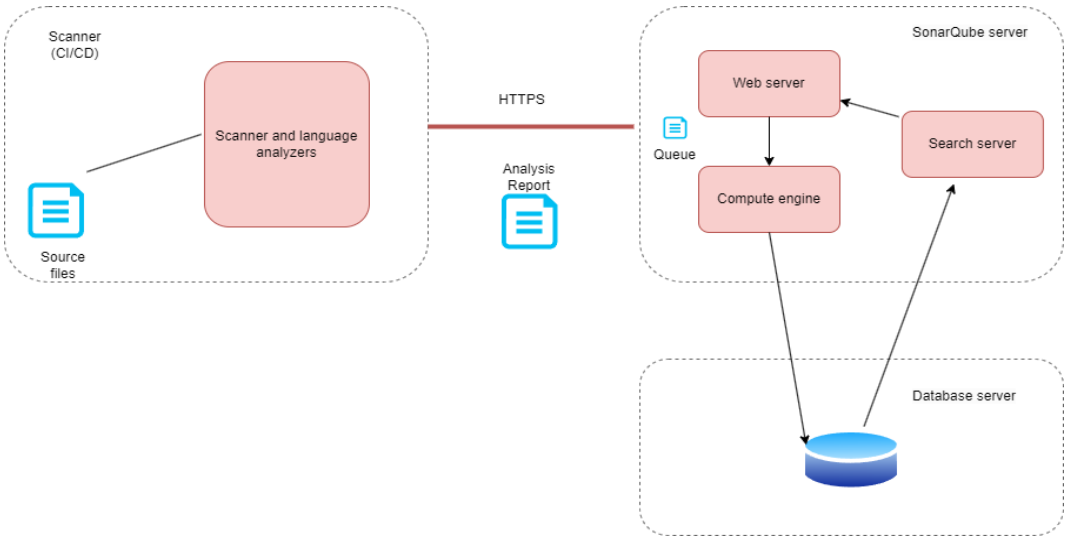

Figure 5.1: SonarQube Server Structure

5.3 Installation

In this project, it was decided to proceed with the server installation rather than using Docker. One key reason for this choice is the need for more granular control over the configuration and management of the SonarQube environment, as shown in Figure 5.1 . The following commands display the commands needed to proceed with the installation:

```
# 1. Download and Install SonarQube
john@doe:~$ sudo apt-get install zip -y
john@doe:~$ sudo wget
   ↪ https://binaries.sonarsource.com/Distribution
                          /sonarqube/sonarqube-9.6.1.59531.zip
john@doe:~$ sudo unzip sonarqube-9.6.1.59531.zip
john@doe:~$ sudo mv sonarqube-9.6.1.59531 sonarqube
john@doe:~$ sudo mv sonarqube /opt/

# 2. Add SonarQube Group and User
john@doe:~$ sudo groupadd sonar
john@doe:~$ sudo useradd -d /opt/sonarqube -g sonar sonar
john@doe:~$ sudo chown sonar:sonar /opt/sonarqube -R

# 3. Configure SonarQube
john@doe:~$ sudo nano /opt/sonarqube/conf/sonar.properties
# Edit with sonar username, password and url
john@doe:~$ sudo nano
   ↪ /opt/sonarqube/bin/linux-x86-64/sonar.sh
# Add sonar user

# 4. Setup Systemd service
john@doe:~$ sudo nano /etc/systemd/system/sonar.service
# Add service configuration of sonar
john@doe:~$ sudo systemctl enable sonar
john@doe:~$ sudo systemctl start sonar
john@doe:~$ sudo systemctl status sonar

# 5. Modify Kernel System Limits
john@doe:~$ sudo nano /etc/sysctl.conf
# Increase limit
john@doe:~$ sudo reboot

# 6. Access SonarQube Web Interface
# Access through http://IP:9000
```

Listing 5.1: SonarQube Installation Commands

5.4 Analysis Result

The SonarQube analysis presented in Figure 5.2 reflects the state of the JMEOS codebase with specific exclusions due to the project's context. The **builder** package, the **utils** package, and the **tutorials** package were deliberately omitted from this analysis. The reasons behind such exclusions stems from the nature of the files within these packages. Some are automatically generated, and others, such as utility scripts or tutorial examples, do not necessitate unit tests due to subsequent role in the project. This approach in the analysis ensures that the metrics accurately represent the core, manually authored code that primarily drives the functionality of JMEOS.

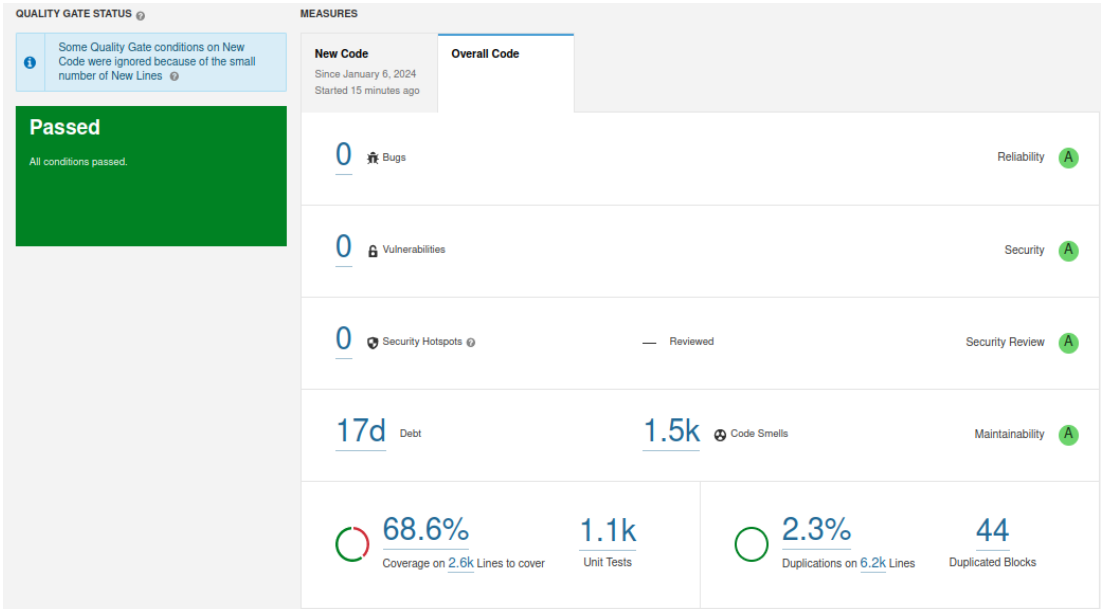

Figure 5.2: SonarQube Analysis Result

We can comment the resulted analysis and say that:

- **Bugs (0)**: Containing no bugs reported is an excellent sign, as it suggests the code is functionally correct and likely to operate as intended without causing errors at runtime, reflecting a mature and robust development process.
- **Vulnerabilities (0)**: A count of zero vulnerabilities is indicative of secure coding practices, suggesting that the code make usage of the last library versions and good practices, thereby reducing the likelihood of exploitation.
- **Security Hotspots (0)**: No security hotspots imply that the code does not contain any sensitive areas that might need a closer security review.
- **Debt (17d)**: An 17-day technical debt points to areas that could benefit from

improvement, but given the complexity and the size of JMEOS, this debt is completely manageable and indicates that the codebase is well-maintained overall.

- **Code Smells (1.5k)**: While 1.5k code smells might seem high, they are often minor issues and not correctness problems. Addressing them can improve code clarity and maintainability, but their presence doesn't necessarily impact the immediate functionality.
- **Coverage (68.6%)**: A test coverage of 68.6% shows a strong commitment to testing, ensuring that a majority of the code has been validated through automated tests.
- **Unit Tests (1,139)**: Having 1,139 unit tests demonstrates a thorough approach to validating independent components of the code, contributing to the overall reliability and ease of maintaining the project.
- **Duplications (2.3%)**: The reasonable rate of code duplication indicates adherence to DRY (Don't Repeat Yourself) principles, promoting code reuse and simplification.
- **Duplicated Blocks (44)**: The presence of 46 duplicated blocks, while not excessive, points to opportunities for further refactoring. These duplication blocks may due to the numerous abstract class which behavior is overloaded through child classes.

Chapter 6

Use Case Examples

6.1 Files Implementations

Incorporating practical examples into JMEOS represent a pivotal aspect of demonstrating its real-world applicability and functional range. By creating three distinct example files, we aim to illuminate the varied uses of JMEOS's types and methods. These examples bridge theoretical concepts with practical application, offering users a tangible understanding of how JMEOS functions in diverse scenarios. These numerous examples are highly inspired by the MEOS examples.

The second section of this chapter will delve into a detailed benchmark analysis. The importance of these use case examples is reflected in their role in assessing performance metrics. By applying JMEOS across different datasets, we can critically assess its efficiency in terms of processing speed. This evaluation is particularly relevant when handling large-scale data, a common challenge in temporal data management. More precisely, this analysis will compare JMEOS's time performance against MEOS and PyMEOS across various data scales for the **read_ais** file.

Such comparative studies are essential to evaluate JMEOS's efficacy in large-scale data processing and to demonstrate its potential in real-time data handling. Such use case examples and benchmark analyses are indispensable in demonstrating JMEOS's practicality and robustness, making it a valuable tool for developers and researchers working with temporal data.

6.1.1 Hello World

This first program, stored in the file **hello_world.java**, is a comprehensive demonstration of using various temporal geometric point types from the library. It begins by initializing MEOS inner library and then creates instances of TGeomPointInst, TGeomPointSeq, and TGeomPointSeqSet. These instances represent diverse temporal geometric types, that we described in detail in Chapter 3, each with their unique interpolation methods. These temporal instances will be transformed as String object in MF-JSON format. Finally, these strings will be concatenated and well-formated with basic strings, before being outputted to the user.

The example exhibits the use of these types in practical scenarios, like tracking geometric points over time with precise temporal markers. The code also emphasizes the handling of different interpolation behaviors - discrete, linear, and stepwise - and demonstrates the conversion of these temporal types into MF-JSON formats.

We can compare both outputs generated by MEOS and JMEOS, which matches perfectly.

```
--------------------
| Temporal Instant |
--------------------

WKT:
----
POINT(1 1)@2000-01-01

MF-JSON:
--------
{
  "type": "MovingGeomPoint",
  "bbox": [
    [
      1,
      1
    ],
    [
      1,
      1
    ]
  ],
  "period": {
    "begin": "2000-01-01T00:00:00+00",
```

Figure 6.1: Hello World program output

6.1.2 Read AIS

The read_ais tutorial file processes AIS data, specifically focusing on maritime tracking. The program reads a CSV file, extracts data like MMSI, latitude, longitude, and speed, and then uses JMEOS classes to produce temporal instances and sets.

The code initializes JMEOS and sets the working directory to read the AIS data file. It then defines an AIS_record class to contain data from each CSV record, such as MMSI, geographic coordinates, and speed. The code employs Scanner type to read the CSV file line by line, parsing each line into tokens to extract the necessary data.

These extracted tokens are used to create corresponding temporal type like TGeogPointInst and TimestampSet. TGeogPointInst is used to represent a point in geographical space at a specific instance in time, while TimestampSet represents a set of discrete temporal points.

The process involves converting the latitude and longitude into a point representation (WKT) and coupling it with a timestamp to create a TGeogPointInst instance. This instance represents the ship's location at a specific time. Identically, speed over ground (SOG) is handled using a TFloatInst object, representing the speed at a specific time.

In conclusion, the program outputs the MMSI, the point and float instant (Location and SOG). Once more, the output obtained can be compared with the same output obtained from MEOS, where both are matching.

6.1.3 Simplify BerlinMOD

The **simplify_berlinmod** file is an example demonstrating the processing and simplification of synthetic trip data from a CSV file. This program is particularly interesting as it focus on geographical trip data simplification.

The program initiates by setting up MEOS, and preparing for file reading. It utilizes a CSV file, **trips.csv**, containing synthetic trip data. Each line in this file encapsulates information like trip ID, vehicle ID, date, sequence number, and trip data in hexadecimal WKB format. These pieces of data are parsed and extracted for processing.

The core of this program performs two types of trip data simplifications – Douglas-Peucker (DP) and Synchronized Euclidean Distance (SED) simplifications. DP simplification aims to reduce the number of points in the trip's geometry, maintaining a balance between data integrity and simplicity, based on a set distance threshold (DELTA_DISTANCE). On the other hand, SED simplification, akin to DP, incorporates the temporal dimension, making it more suited for spatiotemporal data. These simplifications are applied through the temporal_simplify_dp method, which operates on TGeogPointInst types.

Post simplification, the program outputs details for each trip, such as the vehicle ID, date, sequence number, and the count of instants pre and post simplification. This output uses the num_instants method of the TGeogPointInst, described in Chapter 3 of the thesis.

Additionally, the program measures the execution time for processing and simplifying all the trips. This metric is essential to assess JMEOS's efficiency in handling large spatiotemporal datasets.

To structure the trip data, the program employs an inner class, `trip_record`, which holds the trip ID, vehicle ID, date, sequence number, and the temporal trip data. This structuring is vital for organized data manipulation and simplification within the program.

```
Vehicle: 5, Date: 2020-06-01, Seq: 2, No. of instants: 1170, No. of instants DP: 81, No. of instants SED: 535
Vehicle: 5, Date: 2020-06-01, Seq: 3, No. of instants: 292, No. of instants DP: 24, No. of instants SED: 109
Vehicle: 5, Date: 2020-06-01, Seq: 4, No. of instants: 278, No. of instants DP: 24, No. of instants SED: 102
Vehicle: 5, Date: 2020-06-02, Seq: 1, No. of instants: 1266, No. of instants DP: 91, No. of instants SED: 566
Vehicle: 5, Date: 2020-06-02, Seq: 2, No. of instants: 1154, No. of instants DP: 81, No. of instants SED: 526
Vehicle: 5, Date: 2020-06-02, Seq: 3, No. of instants: 184, No. of instants DP: 16, No. of instants SED: 83
Vehicle: 5, Date: 2020-06-02, Seq: 4, No. of instants: 200, No. of instants DP: 15, No. of instants SED: 84
Vehicle: 5, Date: 2020-06-03, Seq: 1, No. of instants: 1207, No. of instants DP: 91, No. of instants SED: 540
Vehicle: 5, Date: 2020-06-03, Seq: 2, No. of instants: 1180, No. of instants DP: 81, No. of instants SED: 536
Vehicle: 5, Date: 2020-06-03, Seq: 3, No. of instants: 110, No. of instants DP: 17, No. of instants SED: 59
Vehicle: 5, Date: 2020-06-03, Seq: 4, No. of instants: 144, No. of instants DP: 18, No. of instants SED: 73
Vehicle: 5, Date: 2020-06-04, Seq: 1, No. of instants: 1255, No. of instants DP: 91, No. of instants SED: 555
Vehicle: 5, Date: 2020-06-04, Seq: 2, No. of instants: 1182, No. of instants DP: 81, No. of instants SED: 531
Vehicle: 5, Date: 2020-06-04, Seq: 3, No. of instants: 261, No. of instants DP: 23, No. of instants SED: 117
Vehicle: 5, Date: 2020-06-04, Seq: 4, No. of instants: 1492, No. of instants DP: 117, No. of instants SED: 675
Vehicle: 5, Date: 2020-06-04, Seq: 5, No. of instants: 723, No. of instants DP: 57, No. of instants SED: 328
Vehicle: 5, Date: 2020-06-04, Seq: 6, No. of instants: 168, No. of instants DP: 16, No. of instants SED: 64
Time taken:
1.060434984
```

Figure 6.2: Simplify BerlinMOD JMEOS program output.

```
55 records read.
0 incomplete records ignored.
Vehicle: 1, Date: 2020-06-01, Seq: 1, No. of instants: 2472, No. of instants DP: 163, No. of instants SED: 1149
Vehicle: 1, Date: 2020-06-01, Seq: 2, No. of instants: 2420, No. of instants DP: 138, No. of instants SED: 1103
Vehicle: 1, Date: 2020-06-02, Seq: 1, No. of instants: 2481, No. of instants DP: 163, No. of instants SED: 1147
...
Vehicle: 5, Date: 2020-06-04, Seq: 3, No. of instants: 99, No. of instants DP: 17, No. of instants SED: 56
Vehicle: 5, Date: 2020-06-04, Seq: 4, No. of instants: 116, No. of instants DP: 18, No. of instants SED: 58
```

Figure 6.3: Simplify MEOS program output

When comparing Figure 6.2 and Figure 6.3, we clearly see that the date related to Vehicle 5 and Seq 3 to 4 are both matching.

6.2 File Benchmarking

As stated in the introduction of this chapter, time performance benchmarking could be a useful metric to compare MEOS, PyMEOS and JMEOS. Due to time and size constraints of the thesis, it was decided to perform the benchmark on the **read_ais** example. This benchmark is more a first sight on the difference of performance between the three libraries.

6.2.1 Dataset

In order to obtain consistent and strong results, it was decided to generate 3 CSV datasets of 3 different scales: 0.2, 0.5 and 1, where scale 1 represents a file with 1 millions rows.

The dataset used for the example is collection of data points typically used in maritime tracking and specifically related to Automatic Identification System (AIS) data, which is used on ships and by vessel traffic services for identifying and locating vessels through the exchange of electronic data with other nearby ships and AIS base stations. Table 6.1 shows the first 3 rows of the scale 0.2 file.

t	MMSI	Latitude	Longitude	SOG
2021-01-08 00:00:00	265513270	57.059	12.272388	0
2021-01-08 00:00:01	219027804	55.94244	11.866278	0
2021-01-08 00:00:01	265513270	57.059	12.272388	0

Table 6.1: Four first rows of the **scale2.csv** file

Each file contains:

- **T**: It represents the timestamp for each data entry. It shows the exact date and time when the data were recorded.
- **MMSI**: Standing for Maritime Mobile Service Identity, it constitutes a unique identifier assigned to a vessel.
- **Latitude**: Define the geographical coordinate that specifies the north–south position of a point on the Earth's surface and measured in degrees.
- **Longitude**: Representing the geographical coordinate that specifies the east-west position of a point on Earth and measured in degrees.
- **SOG**: Standing for Speed Over Ground and define the speed at which a boat or a vessel is moving, over the surface of the water, measures in knots.

6.2.2 System Used

Tests were executed in a sterile environment. This was achieved by initiating the procedures immediately following the system's boot-up, ensuring a pure state with maximal available RAM. Such environments guarantee the absence of extra unuseful processes by providing spotless and accurate performance metrics. The specification of the computer performing the benchmark is depicted in the Table 6.2:

6.2.3 Time Performance

The first and most obvious measurement performed is related to time performance. More precisely, each of the three file were executed 5 times on JMEOS, MEOS and PyMEOS. The result obtained is an average value over these 5 iterations. The Figure 6.4 depicts the outputs of the differents runs. As we can observe, JMEOS takes on average 8.34 seconds to process the file of scale 0.2, 17.93 seconds for the scale 0.5 and 33.06 seconds for the file with one millions rows. When compared to MEOS library, we can pinpoint that MEOS processes the file 4 to 5 times faster than JMEOS. This

Specification	**Details**
Processor	Intel Core i3-6100U, 2.30 GHz
Cores/Threads	2 cores, 4 threads
Cache	3 MB
Graphics	Intel HD Graphics 520
Memory	8 GB RAM, DDR4, 2133 MHz
Storage	256 GB SSD
Display	12.5-inch, 1366x768 resolution
Battery Life	Up to 13.04 hours
Operating System	Windows 10 Pro 64
Ports	HDMI, USB 3.0, OneLink+, RJ45, MiniDP, etc.
Wireless	Intel Dual Band Wireless-AC 8260, Bluetooth 4.1

Table 6.2: Benchmark Computer Specifications

is completely coherent since, MEOS is written in a low-level language that manipulates the memory directly and so handle more carefully the data. Java source code is first compiled by the Java compiler into bytecode, which is a form of intermediate code. This bytecode is not machine-specific and can be executed on any machine that has a Java Virtual Machine [34] . In addition to that, the dependencies that we added may slow down the whole process, since it stacks up more linkage and data to synchronize. Despite these differences, this speedup factor seems to be relatively stable over the scaling factors. PyMEOS time performance is poor when compared to JMEOS. On the 0.2 scale factor, the time taken is relatively high with nearly 13 seconds. This dynamic is the same for the others scale factors, where PyMEOS process the same file with 35-50% time difference compared to JMEOS. We can assume that this is due to the fact that Python is a high-level language, offering easy coding standards which leads to higher compiling time. Moreover, PyMEOS make usage of more dependencies than JMEOS, which again, leads to more linkage time. Another hypothesis to take into account for these differences is the implementation strategy. Sometimes using some data types or blocks leads to a negative impact in the overall performance. This difference with PyMEOS underlines the good choice made from the beginning of the development of JMEOS, by minimizing the number of dependencies, choosing an efficiently low level CFFI, and by using correctly the types and functions made at our disposal.

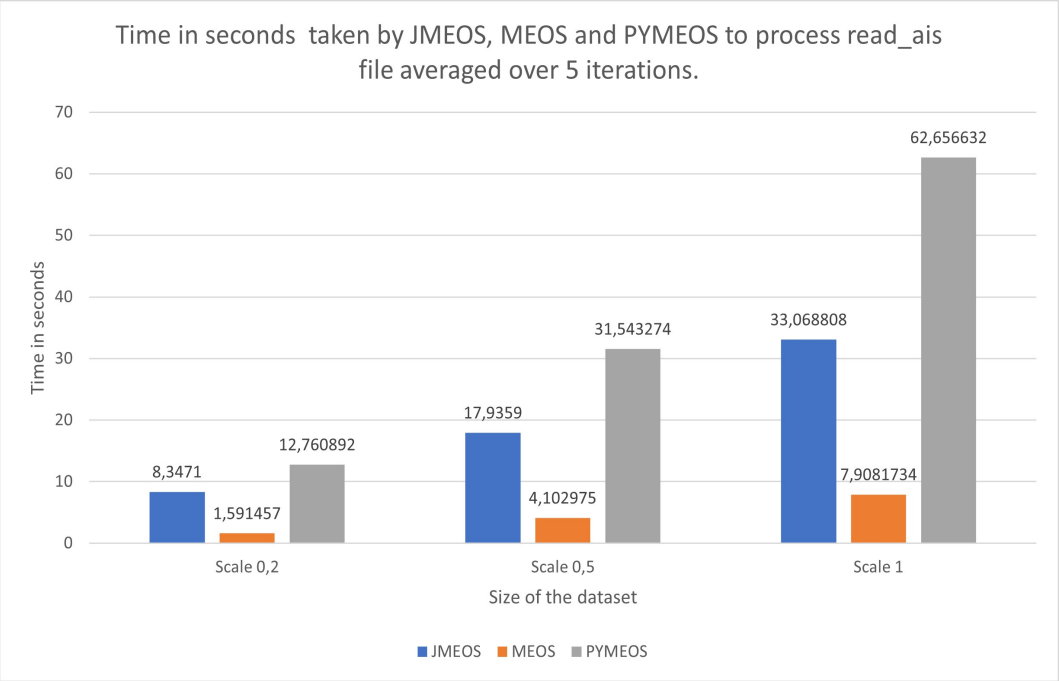

Figure 6.4: Time taken in seconds by JMEOS, MEOS and PYMEOS to process **read_ais** file.

6.2.4 Troughput efficiency

Thanks to the last observation, we can define a metric which is the throughput efficiency, defined by:

$$TE = R/T$$

, where TE is expressed in rows per seconds, R is the number of rows contained in the file and T the time taken to process the file. The TE for each library is shown in Figure 6.5. As we can see, the efficiency is clearly increasing when the scale size augments. This may due to the fact that the JVM smartly learns and store some metadata information about the processing of the file in order to optimize the runs when larger data come in. The efficiency of MEOS seems to be dangling by increasing and decreasing over time but staying in an average level of 120000 rows per second. This small decrease may be due to an unusual iteration that lowered the entire efficiency. Concerning PyMEOS, it is clear that it stagnate under 20000 rows per second with always the same level. It seems like this efficiency does not change any matter the scaling size, meaning that Python correctly manages the size malleability.

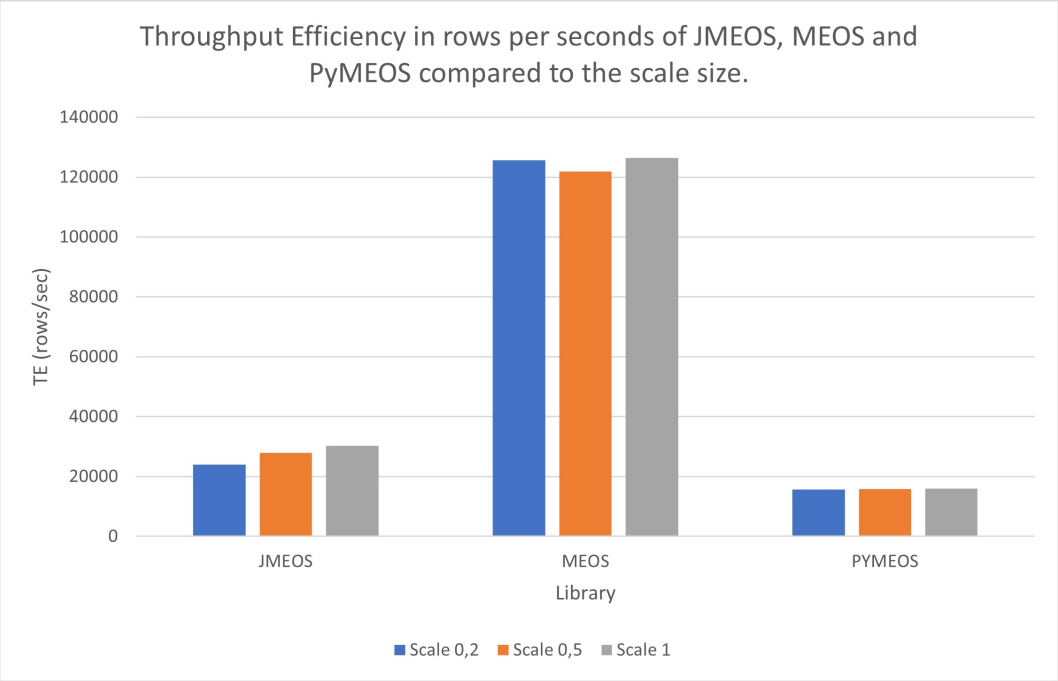

Figure 6.5: Throughput efficiency in rows per seconds after processing **read_ais** file.

Chapter 7

Future Work

1. **Error handling**: It would be pertinent to address the aspect of error handling. JNR-FFI, while offering significant advantages in terms of performance and ease of use, comes with certain limitations in terms of documentation and debuggability. This is particularly relevant when compared to other C Foreign Function Interfaces (CFFI).

   Enhancing error handling in JMEOS could substantially aid in debugging, especially for issues related to interfacing with the underlying C library through JNR-FFI. Currently, JNR-FFI's less extensive documentation and intrinsic complexity in debugging C integration points can pose challenges for developers. Improved error handling mechanisms could provide more informative and precise feedback on issues that occur at the boundary between Java and native C code. This would not only facilitate more rapid resolution of problems but also make JMEOS more robust and user-friendly.

2. **Implementation of new tests**: It is indeed crucial to discuss the enhancement of test coverage. While the current scope of the thesis allowed for substantial development and testing, achieving 100% test coverage was not feasible within the time constraints. However, aiming for complete test coverage in future iterations of JMEOS is thoroughly recommended and would significantly bolster the reliability of the software.

3. **Implementation of last methods**: It is substantial to acknowledge that the current iteration of JMEOS represents a foundational version of the library. While it successfully implements a substantial range of functionalities, it is also essential to note that a strictly limited number of methods are still pending implementation. Completing these will be crucial for unlocking the maximum potential of JMEOS.

   The inclusion of these remaining methods in forthcoming versions would not only complete the library's capabilities but also enhance its applicability for a broader range of use cases. This approach will ensure that JMEOS fully encapsulates the functionality of its underlying MEOS C library, thereby offering a more robust and versatile toolset for Java developers working with spatiotemporal data.

4. **Adding new examples**: Another aspect to consider for the future work, is the implementation of additional example files that utilize real-world data. These examples would serve as practical demonstrations of JMEOS functionalities in realistic scenarios, providing valuable insights for users into how the library can be applied effectively in various contexts. Focusing on creating a diverse set of example files covering a range of applications and data sets would be immensely

beneficial. These could include, but are not limited to, examples in urban planning, environmental monitoring, transportation logistics, and geographic information system (GIS) applications.

5. **Creation of new MEOS bindings**: The expansion of MobilityDB's (through MEOS) by developing new bindings for additional programming languages such as C and JavaScript could be interesting. This expansion would not only cater to a broader developer community but also enable diverse applications across different platforms and environments. It would enable developers across various language ecosystems to leverage MobilityDB's powerful spatiotemporal data processing capabilities, fostering a more inclusive and diverse user base. Moreover, the comparative analysis of different bindings would contribute valuable knowledge to the field of spatiotemporal data processing, guiding future developments in this rapidly evolving domain.

Chapter 8

Conclusion

In conclusion, this thesis presents and thoroughly describe JMEOS, a first Java binding approach of the MEOS library. The introduction discusses about the current technologies to store and manipulates moving objects. It presents the MobilityDB open-source solution alongside with the adapters and programming language. Chapter 2, lists the system requirements to build and execute JMEOS are clearly defined. Moreover the CFFI choice for JNR-FFI, mainly motivated by time efficiency and ease of use, is explicitly detailed.

In, Chapter 3 the overall design of the binding is depicted. More specifically, the wrapping process of MEOS API functions is performed in two separate steps, the functions signature extraction done through regex, and the functions generation using `StringBuilders`. Next, the `TBox` and `STBox` classes implementation are detailed with the set of applicable functions such as conversion, position or topological operations. Concrete types belonging to the collection package are all inheriting from `Span`, `Spanset` or `Set` base classes due to their common behaviors. Among these concrete types, `Period`, `PeriodSet`, `TimestampSet`, as well as the Float and Int are the ones that needed a particular attention due to the fact that they contains many operations such that transformations, topological, position and set operations allowing a rich and strong temporal type manipulation for real-case usage. Finally, temporal types, which are the core of the project, are focused on the main abstract `Temporal` types with its three inheriting subtypes: `TInstant`, `TSequence` and `TSequenceSet`. `TBool`, `TInt`, `TFloat` and `TText` concrete classes all implements an interface that regroups a set of applicable functions and inherits from the 3 subtypes. `TPoint` is differently implemented with indirect inheritance and indirect implementation of interfaces. Both `TGeomPoint` and `TGeogPoint` are among the most interesting and valuable types due to their nature and set of attributes. Lastly, a dockerization process of JMEOS, enhancing the portability and distribution of the library is implemented.

The Chapter 4, describes the motivation and an overview of the implemented unit tests alongside with the JUnit execution. Chapter 5 discusses about the high standard code quality of JMEOS that contains no vulnerabilities, no bugs and that implements more than 1,100 unit tests.

In Chapter 6, execution of use case tests with real data are performed. Three different files are positively tested and compared with MEOS results. The second part of the chapter compares the time performance between JMEOS, MEOS and PyMEOS where it pinpoint the fact that JMEOS performs worse than MEOS but better than PyMEOS due to the strategic choices made in the beginning of the thesis.

Chapter 7, provides possible future work such as test and examples improvements, but also error handling and an opening to new bindings.

References

[1] Md Mahbub Alam, Luis Torgo, and Albert Bifet. "A survey on spatio-temporal data analytics systems". In: *ACM Computing Surveys* 54.10s (2022), pp. 1–38.

[2] Charalampos Arapidis. *Sonar Code Quality Testing Essentials*. Packt Publishing, 2012. ISBN: 9781849517867.

[3] Samuel Audet, Ryan Osial, and Alexey Rochev. *JavaCPP*. https://github.com/bytedeco/javacpp. Accessed: 24/11/2023. 2019.

[4] A. N. Author. "Advancements in Geospatial Data Management: The Role of PostGIS". In: *Journal of Geospatial Technology* 10.2 (2023), pp. 101–120.

[5] Mohamed Bakli, Esteban Zimányi, Arthur Lesuisse, Maxime Schoemans, and Krishna Chaitanya. *MobilityDB-python*. https://github.com/MobilityDB/MobilityDB-python. Accessed: 10/11/2023. 2021.

[6] Bence Barta, Günter Manz, István Siket, and Rudolf Ferenc. "Challenges of SonarQube plug-in maintenance". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, pp. 574–578.

[7] David Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[8] Joshua Bloch. *Effective Java*. 3rd. Boston, MA, USA: Addison-Wesley Professional, 2018. ISBN: 978-0134685991. URL: https://www.pearson.com/en-us/subject-catalog/p/Bloch-Effective-Java-3rd-Edition/P200000000138/9780134686042.

[9] Krishna Chaitanya. *jmeos*. https://github.com/adonmo/jmeos. Accessed: 24/11/2023. 2021.

[10] Theo Combe, Antony Martin, and Roberto Di Pietro. "To docker or not to docker: A security perspective". In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62.

[11] Pär Emanuelsson and Ulf Nilsson. "A comparative study of industrial static analysis tools". In: *Electronic notes in theoretical computer science* 217 (2008), pp. 5–21.

[12] Zimányi Esteban, Jose Antonio Lorencio Abril, Mohamed Bakli, Ajouaou Soufiane, and Mahmoud Sakr. *MobilityDB-BerlinMOD*. https://github.com/MobilityDB/MobilityDB-BerlinMOD. Accessed: 10/11/2023. 2023.

[13] Zimányi Esteban, Vicky Vergara, Mohamed Bakli, Arthur Lesuisse, Maxime Schoemans, Mahmoud Sakr, Robin Choquet, and Krishna Chaitanya. *MEOS in MobilityDB*. https://github.com/MobilityDB/MobilityDB/tree/master/meos. Accessed: 10/11/2023. 2023.

[14] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 2nd. Boston, MA, USA: Addison-Wesley Professional, 2018. ISBN: 9780134757599. URL: https://www.pearson.com/en-us/subject-catalog/p/refactoring-improving-the-design-of-existing-code/P200000000254/9780134757704.

[15] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. 1st ed. Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2009, p. 384. ISBN: 978-0-321-50362-6.

[16] "Generating and Accessing Javadoc". In: *Pro NetBeans™ IDE 5.5 Enterprise Edition*. Berkeley, CA: Apress, 2007, pp. 319–331. ISBN: 978-1-4302-0381-0. DOI: 10.1007/978-1-4302-0381-0_12. URL: https://doi.org/10.1007/978-1-4302-0381-0_12.

[17] Michael F. Goodchild and Donna J. Peuquet. "Spatio-Temporal Data Handling and Visualization in GIS". In: *Annals of GIS* (2021).

[18] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification, Java SE 14 Edition*. Oracle America, Inc., 2020. URL: https://docs.oracle.com/javase/specs/jls/se14/html/index.html.

[19] Joe Green. "PostgreSQL: When open source gets serious". In: *TechHQ* (Aug. 2019). URL: https://techhq.com/2019/08/postgres-enterprise-professional-database/.

[20] Andrew Habib and Michael Pradel. "How many of all bugs do we find? a study of static bug detectors". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 317–328.

[21] Michal Kvet, Emil Kršák, and Karol Matiaško. "Study on Effective Temporal Data Retrieval Leveraging Complex Indexed Architecture". In: *Applied Sciences* 11.3 (2021). ISSN: 2076-3417. DOI: 10.3390/app11030916. URL: https://www.mdpi.com/2076-3417/11/3/916.

[22] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux Journal* 2014.239 (2014). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[23] Bruce Momjian. *PostgreSQL: Introduction and Concepts*. 1st ed. Addison-Wesley New York, 2001, p. 462. ISBN: 0201703319. URL: https://www.abebooks.com/9780201703313/PostgreSQL-Introduction-Concepts-Momjian-Bruce-0201703319/plp.

[24] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd ed. Wiley, 2011, p. 256. ISBN: 978-1-118-03196-4.

[25] William R Nichols Jr. "The Cost and Benefits of Static Analysis During Development". In: *arXiv preprint arXiv:2003.03001* (Mar. 2020). DOI: 10.48550/arXiv.2003.03001. URL: https://arxiv.org/abs/2003.03001.

[26] Regina O. Obe and Leo S. Hsu. *PostGIS in Action*. Manning Publications, 2011, p. 520. ISBN: 9781935182269. URL: https://www.oreilly.com/library/view/postgis-in-action/9781935182269/.

[27] Patroklos Papapetrou. *SonarQube in Action*. Manning, 2013. ISBN: 9781617290954.

[28] Andrew Pavlo and Matthew Aslett. "What's Really New with NewSQL?" In: *ACM Sigmod Record* 45.2 (2016), pp. 45–55. DOI: 10.1145/3003665.3003674. URL: https://dblp.org/rec/journals/sigmod/PavloA16.

[29] George Percivall. "OpenGIS international standards for GEOSS interoperability arrangements". In: *2006 IEEE International Symposium on Geoscience and Remote Sensing*. IEEE. 2006, pp. 2481–2484.

[30] Tatiana del Pilar Millan Poveda. *MobilityDB-JDBC*. https://github.com/MobilityDB/MobilityDB-JDBC. Accessed: 10/11/2023. 2022.

[31] Enrico Pirozzi, Ibrar Ahmed, and Gregory Smith. *PostgreSQL 10 High Performance: Expert techniques for query optimization, high availability, and efficient database maintenance*. Packt Publishing Ltd, 2018. ISBN: 9781788472456. URL: https://www.oreilly.com/library/view/postgresql-10-high/9781788474924/.

[32] Nigel Poulton. *Docker Deep Dive: 2023 Edition*. Publishdrive, 2023. ISBN: 97819165-85256.

[33] Ioannis Samoladas, Ioannis Stamelos, Lefteris Angelis, and Apostolos Oikonomou. "Open source software development should strive for even greater code maintainability". In: *Communications of the ACM* 47.10 (2004), pp. 83–87.

[34] Herbert Schildt. *Java: A Beginner's Guide*. Provides an overview of Java programming, including its principles, how Java code is compiled and executed, and the role of JVM in achieving platform independence. McGraw-Hill Education, 2014. ISBN: 978-0071809252.

[35] Gholam Ali Shafabakhsh, Afshin Famili, and Mohammad Sadegh Bahadori. "GIS-based spatial analysis of urban traffic accidents: Case study in Mashhad, Iran". In: *Journal of traffic and transportation engineering (English edition)* 4.3 (2017), pp. 290–299.

[36] Michael Stonebraker. "SQL databases v. NoSQL databases". In: *Communications of the ACM* 53.4 (2010), pp. 10–11.

[37] Christian Strobl. "PostGIS". In: *Encyclopedia of GIS*. Ed. by Shashi Shekhar and Hui Xiong. Boston, MA: Springer US, 2008, pp. 891–898. ISBN: 978-0-387-35973-1. DOI: 10.1007/978-0-387-35973-1_1012. URL: https://doi.org/10.1007/978-0-387-35973-1_1012.

[38] Patrick Thomson. "Static Analysis". In: *Communications of the ACM* 65.1 (2022), pp. 50–54. DOI: 10.1145/3486592.

[39] Enmei Tu, Guanghao Zhang, Lily Rachmawati, Eshan Rajabally, and Guang-Bin Huang. "Exploiting AIS data for intelligent maritime navigation: A comprehensive survey from data to methodology". In: *IEEE Transactions on Intelligent Transportation Systems* 19.5 (2017), pp. 1559–1582.

[40] Guido Van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., 2003, p. 144. ISBN: 9780954161781.

[41] Carlo Zaniolo. *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. Ed. by Carlo Zaniolo. Vol. 15. SIGMOD Record 2. Conference held in Washington, D.C., May 28-30, 1986. New York, NY: ACM Press, 1986, pp. xi, 407. ISBN: 978-0-89791-191-7. URL: https://dblp.org/rec/conf/sigmod/86.

[42] Esteban Zimányi. *Chapter 2. Set and Span Types*. MobilityDB Documentation. Accessed: 24/11/2023. 2021. URL: https://docs.mobilitydb.com/MobilityDB/develop/ch02.html.

[43] Esteban Zimányi. *Enabling Mobility Data in Multiple Computing Environments*. Presented at ICDE 2023, Anaheim, CA, USA. Accessed: 10/11/2023. Mar. 2023. URL: https://docs.mobilitydb.com/pub/DASC_2023_MobilityDB_MEOS.pdf.

[44] Esteban Zimányi. *PyMEOS: A Python binding to the MEOS C Library*. Online. Accessed: 10/11/2023. 2022. URL: https://docs.mobilitydb.com/pub/PyMEOS-OGC-2022-October-slides.pdf.

[45] Esteban Zimányi, Mahmoud Sakr, and Mohamed Bakli. *MobilityDB*. GitHub repository. Accessed: 10/11/2023. 2020. URL: https://github.com/MobilityDB/MobilityDB/wiki/Building-MobilityDB-and-MEOS.

[46] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. "MobilityDB: A mobility database based on PostgreSQL and PostGIS". In: *ACM Transactions on Database Systems (TODS)* 45.4 (2020), pp. 1–42.

[47] Mayra Zurbarán, Thomas Kraft, Stephen Vincent Mather, Bborie Park, and Pedro Wightman. *PostGIS Cookbook: Store, organize, manipulate, and analyze spatial data*. 2nd ed. Packt Publishing Ltd, 2018. ISBN: 9781788299329. URL: https://www.oreilly.com/library/view/postgis-cookbook-second/9781788299329/.

Appendix A

Docker Image

```
FROM debian:bookworm-slim
MAINTAINER nidhalmareghni8@gmail.com

# install corretto after verifying that the key is the one we
    expect.
RUN apt-get update \
  && apt-get install -y git curl gnupg build-essential tree vim
     cmake postgresql-server-dev-15 libproj-dev libjson-c-dev
     libgsl-dev libgeos-dev postgis \
  && export GNUPGHOME="$(mktemp -d)" \
  && curl -fL https://apt.corretto.aws/corretto.key | gpg --batch
     --import \
  && gpg --batch --export
     '6DC3636DAE534049C8B94623A122542AB04F24E3' >
     /usr/share/keyrings/corretto.gpg \
  && rm -r "$GNUPGHOME" \
  && unset GNUPGHOME \
  && echo "deb [signed-by=/usr/share/keyrings/corretto.gpg]
     https://apt.corretto.aws stable main" >
     /etc/apt/sources.list.d/corretto.list \
  && apt-get update \
  && apt-get install -y java-21-amazon-corretto-jdk

# BUILD MobilityDB
RUN git clone https://github.com/MobilityDB/MobilityDB.git -b
    develop /usr/local/src/MobilityDB
RUN mkdir -p /usr/local/src/MobilityDB/build
RUN cd /usr/local/src/MobilityDB/build && \
    cmake -DMEOS=ON .. && \
    make -j$(nproc) && \
    make install

# COMMON FOR ALL IMAGES
ENV MAVEN_HOME /usr/share/maven
ENV MAVEN_CONFIG "/root/.m2"
```

Listing A.1: Docker Image

```
1
2  COPY --from=maven:3.9.6-eclipse-temurin-11 ${MAVEN_HOME}
       ${MAVEN_HOME}
3  COPY --from=maven:3.9.6-eclipse-temurin-11
       /usr/local/bin/mvn-entrypoint.sh
       /usr/local/bin/mvn-entrypoint.sh
4  COPY --from=maven:3.9.6-eclipse-temurin-11
       /usr/share/maven/ref/settings-docker.xml
       /usr/share/maven/ref/settings-docker.xml
5
6  RUN ln -s ${MAVEN_HOME}/bin/mvn /usr/bin/mvn
7
8  # ADD PROJECT MobilityDB-JMEOS
9  RUN git clone https://github.com/nmareghn/MobilityDB-JMEOS
       /usr/local/jmeos
10 RUN rm /usr/local/jmeos/src/main/resources/lib/libmeos.so
11 RUN cp /usr/local/lib/libmeos.so
       /usr/local/jmeos/src/main/resources/lib/libmeos.so
12 RUN rm /usr/local/jmeos/jar/*
13
14 # CLEAN_UP
15 RUN rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*lists/*
16 RUN apt-get remove --purge --autoremove -y curl gnupg
17
18 # END_POINT
19 WORKDIR /usr/local/jmeos
20 CMD ["/bin/bash"]"
```

Listing A.2: Docker Image Continuation

Appendix B

Methods Implemented in Time and Box Package

Method	**STBox**	**TBox**	**Period**	**PeriodSet**	**TimestampSet**
_get_box	X				
add	X	X	X	X	X
contains	X	X	X	X	X
copy	X	X	X	X	X
distance			X		
duration			X		
duration_in_second			X		
end_element					X
end_period				X	
end_span				X	
end_timestamp				X	
eq	X	X	X	X	X
expand		X	X		
expand_numerical	X				
expand_stbox	X				
from_expanding_bounding_box_geom	X				
from_expanding_bounding_box_tpoint	X				
from_geometry	X				
from_geometry_datetime	X				
from_geometry_period	X				
from_hexwkb	X	X	X	X	X
from_space_datetime	X				
from_space_period	X				
from_time	X	X			
from_tnumber		X			
from_tpoint	X				
from_value_number		X			
from_value_span		X			

Method	**STBox**	**TBox**	**Period**	**PeriodSet**	**TimestampSet**
from_value_time		X			
geodetic	X				
getSrid	X				
getTMax	X				
getTMin	X				
get_inner	X	X	X	X	X
get_space	X				
get_tmax_inc	X	X			
get_tmin_inc	X	X			
greaterThan	X	X	X	X	X
greaterThanOrEqual	X	X	X	X	X
has_t	X	X			
has_x		X			
has_xy	X				
has_z	X				
hash			X	X	X
intersection	X	X	X	X	X
isGeodetic	X				
is_above	X				
is_adjacent	X		X	X	X
is_adjacent_tbox		X			
is_after	X	X	X	X	X
is_before	X	X	X	X	X
is_behind	X				
is_below	X				
is_contained_in	X	X	X	X	X
is_front	X				
is_left	X	X			
is_over_or_above	X				
is_over_or_after	X	X	X	X	X
is_over_or_before	X	X	X	X	X
is_over_or_behind	X				
is_over_or_below	X				
is_over_or_front	X				
is_over_or_left	X	X			
is_over_or_right	X	X			

Method	**STBox**	**TBox**	**Period**	**PeriodSet**	**TimestampSet**
is_right	X	X			
is_same	X	X	X	X	X
lessThan	X	X	X	X	X
lessThanOrEqual	X	X	X	X	X
lower			X		
lower_inc			X		
minus			X	X	X
mul	X	X	X	X	X
nearest_approach_distance		X			
nearest_approach_distance_geom	X				
nearest_approach_distance_stbox	X				
nearest_approach_distance_tpoint	X				
notEquals	X	X	X	X	X
num_periods				X	
num_timestamps				X	X
overlaps	X	X	X	X	X
round	X	X			
set_srid	X				
srid	X				
start_element					X
start_period				X	
start_span				X	
start_timestamp				X	
sub			X	X	X
timestamp_n				X	
tmax	X				
tmin	X				
toString	X	X	X	X	X
to_floatspan		X			
to_geometry	X				
to_period	X	X		X	X
to_periodset			X		X
to_span				X	X
to_spanset			X		
union	X	X	X	X	X
upper			X		

Method	**STBox**	**TBox**	**Period**	**PeriodSet**	**TimestampSet**
upper_inc			X		
xmax	X				
xmin	X				
ymax	X				
ymin	X				
zmax	X				
zmin	X				

Table B.1: List of methods names implemented (marked by X) in `TBox`, `STBox`, `Period`, `PeriodSet` and `TimestampSet`

Appendix C

Methods Implemented in Geo, Text and Number Package

Method	**TextSet**	**GeoSet**	**FloatSet**	**FloatSpan**	**FloatSpanSet**	**IntSet**	**IntSpan**	**IntSpanSet**
as_ewkt		X						
as_hexwkb	X							
as_text		X						
as_wkt		X						
contains	X	X	X	X	X	X	X	X
copy							X	
distance			X	X	X	X	X	X
element_n	X		X			X		
elements			X			X		
end_element	X	X	X			X		
end_span					X			X
factory		X						
from_hexwkb							X	
from_wkb							X	
get_inner	X	X	X	X	X	X	X	X
intersection	X		X	X	X	X		X
intersection_geom		X						
intersection_geoset		X						
is_adjacent				X	X		X	X
is_left			X	X	X	X	X	X
is_over_or_left			X	X	X	X	X	X
is_over_or_right			X	X	X	X	X	X
is_right			X	X	X	X	X	X
is_same				X	X		X	X
lower				X			X	
lowercase	X							
minus	X	X	X	X	X	X	X	X
round		X						

Method	**TextSet**	**GeoSet**	**FloatSet**	**FloatSpan**	**FloatSpanSet**	**IntSet**	**IntSpan**	**IntSpanSet**
scale			X	X	X	X		X
shift			X	X		X		X
shift_scale			X	X	X	X		X
span_n					X			X
srid		X						
start_element	X	X	X			X		
start_span					X			X
subtract_from			X			X		
toString	X	X	X	X	X	X	X	X
to_floatset						X		
to_floatspanset								X
to_intset			X					
to_intspan				X				
to_intspanset					X			
to_span			X		X	X		X
to_spanset			X	X		X	X	
tofloatspan							X	
union	X	X	X	X	X	X	X	X
upper				X			X	
uppercase	X							
width				X	X		X	X

Table C.1: List of methods names implemented (marked by X) in `TextSet`, `GeoSet`, `FloatSet`, `FloatSpan`, `FloatSpanSet`, `IntSet`, `IntSpan`, `IntSpanSet`

Appendix D

Methods Implemented in Temporal and Basic Package

Method	**Temporal**	**TBool**	**TFloat**	**TInt**	**TText**	**TPoint**	**TGeogPoint**	**TGeomPoint**
_factory	X							X
always_eq		X						
always_equal	X		X	X	X		X	X
always_greater	X		X	X	X			
always_greater_or_equal	X		X	X	X			
always_less	X		X	X	X			
always_less_or_equal	X		X	X	X			
always_not_equal	X		X	X	X		X	X
angular_difference	X					X		
append_sequence	X							X
as_ewkt	X					X		
as_geojson	X					X		
as_mfjson	X							X
as_wkt	X		X	X	X	X		
at	X					X		X
at_bool		X						
at_max	X							X
at_min	X							X
azimuth	X					X		
bearing_point	X					X		
bounding_box	X							X
bounding_box_point	X					X		
buildTemporalValue	X							
contains	X							
copy	X							X
cumulative_length	X					X		
direction	X					X		
disjoint	X					X		

Method	**Temporal**	**TBool**	**TFloat**	**TInt**	**TText**	**TPoint**	**TGeogPoint**	**TGeomPoint**
distance	X					X		
dyntimewarp_distance	X							X
end_instant	X							X
end_timestamp	X							X
end_value	X	X	X	X	X	X		
eq	X							X
ever_eq		X						
ever_equal	X		X	X	X		X	X
ever_greater	X		X	X	X			
ever_greater_or_equal	X		X	X	X			
ever_intersects	X					X		
ever_less	X		X	X	X			
ever_less_or_equal	X		X	X	X			
ever_not_equal	X		X	X	X		X	X
ever_touches	X					X		
expand	X					X		
frechet_distance	X							X
from_base_temporal	X	X	X	X	X		X	X
from_base_time	X	X	X	X	X		X	X
from_hexwkb	X							X
from_mfjson	X							X
from_mfjson								X
greaterThan	X							X
greaterThanOrEqual	X							X
has_z	X					X		
hash	X							X
hausdorff_distance	X							X
insert	X							
instant_n	X							X
interpolation	X							X
intersects						X		
is_above						X		
is_adjacent	X							X
is_after	X							X
is_before	X							X
is_behind						X		

Method	**Temporal**	**TBool**	**TFloat**	**TInt**	**TText**	**TPoint**	**TGeogPoint**	**TGeomPoint**
is_below						X		
is_contained_in	X							X
is_ever_contained_in						X		
is_ever_disjoint						X		
is_ever_within$_distance$						X		
is_front						X		
is_left						X		
is_over_or_above						X		
is_over_or_after	X							X
is_over_or_before	X							X
is_over_or_behind						X		
is_over_or_below						X		
is_over_or_front						X		
is_over_or_left						X		
is_over_or_right						X		
is_same	X							X
is_simple						X		
is_spatially_contained_in						X		
is_temporally_adjacent	X							X
is_temporally_contained_in	X							X
length						X		
lessThan	X							X
lessThanOrEqual	X							X
max_instant	X							X
max_value			X	X	X			
min_instant	X							X
min_value			X	X	X			
minus	X					X		X
minus_bool		X						
minus_max	X							X
minus_min	X							X
nearest_approach_distance						X		
nearest_approach_instant						X		
never_eq		X						
never_equal			X	X	X		X	X
never_greater			X	X	X			

Method	**Temporal**	**TBool**	**TFloat**	**TInt**	**TText**	**TPoint**	**TGeogPoint**	**TGeomPoint**
never_greater_or_equal			X	X	X			
never_less			X	X	X			
never_less_or_equal			X	X	X			
never_not_equal			X	X	X		X	X
notEquals	X							X
num_instants	X							X
num_timestamps	X							X
overlaps	X							X
period	X							X
round			X			X		
set_interpolation	X							X
set_srid						X		
speed						X		
srid						X		
start_instant	X							X
start_timestamp	X							X
start_value		X	X	X	X	X		X
temporal_and		X						
temporal_and_bool		X						
temporal_equal							X	X
temporal_equal_bool		X						
temporal_equal_number			X	X				
temporal_equal_string					X			
temporal_greater_number			X	X				
temporal_greater_or_equal_string					X			
temporal_greater_or_equal_number			X	X				
temporal_greater_string					X			
temporal_less_number			X					
temporal_less_or_equal_number			X					
temporal_less_or_equal_string					X			
temporal_less_string					X			
temporal_not		X						
temporal_not_equal							X	X
temporal_not_equal_bool		X						
temporal_not_equal_number			X	X				
temporal_not_equal_string					X			

Method	**Temporal**	**TBool**	**TFloat**	**TInt**	**TText**	**TPoint**	**TGeogPoint**	**TGeomPoint**
temporal_or		X						
temporal_or_bool		X						
temporally_contains	X							X
temporally_overlaps	X							X
time	X							X
time_weighted_centroid						X		
timespan	X							X
to_degrees			X					
to_floatrange			X					
to_geometric							X	
to_geographic								X
to_instant	X							X
to_intspan				X				X
to_radians			X					
to_sequence	X							X
to_sequenceset	X							X
to_shapely_geometry						X		
to_string	X	X	X	X	X	X		
to_tint			X					
to_tfloat				X				
touches						X		
update	X							X
value_span			X	X				
value_spans			X	X				
when_false		X						
when_true		X						
within_distance						X		
x						X		
y						X		
z						X		

Table D.1: List of methods names implemented (marked by X) in `Temporal`, `TBool`, `TFloat`, `TInt`, `TText`, `TPoint`, `TGeogPoint`, `TGeomPoint`