Université Libre de Bruxelles

Preparatory work for the Master's thesis

Manipulating Mobility Data in C#

Author : Dudziak Thomas

Promotor : Prof. Zimányi Esteban

Academic year : 2022-2023

August 11, 2023

List of Figures

List of Tables

List of Listings

Contents

Introduction

In a world where everything and everyone is connected, the ability to handle spatio-temporal data has become a fundamental concept of modern society. Spatio-temporal data can be used in countless domains. From wildlife migration to disease spread in biology, from monitoring traffic in high-density areas to controlling entire railway networks, our world can be represented by such kind of data. Lots of the data generated on earth and beyond needs to be modeled with interaction and flexibility between time and space. As we strive for more efficient ways to store and process such data, modern technologies like C# and MobilityDB are tools that provide a real value in our efforts.

Imagine an extremely dense city, where transports are critical for millions of travellers every day. Nowadays, such transportation networks require efficient and robust systems that can respond quickly to the demand to ensure a reliable and high-quality service to the citizens. In dense networks where data flow is almost continuous, we need highly effective algorithms but also compact solutions to store such amounts of data. Obviously, spatio-temporal data is a real challenge - a challenge that MobilityDB aims to resolve.

Among the plethora of technologies that exist, C# resides as a fast, reliable and versatile programming language which embarks the, rich, .NET ecosystem. Providing the ability to manage spatio-temporal data using C# opens a gateway to a myriad of applications of all kinds. Undoubtedly, C# is a multi-purpose and a reference programming language among the developers all across the world.

In this thesis, we delve into the concept of spatio-temporal data and how MobilityDB handles such data. Next, we explore C# and the world of .NET technologies and its flexibility. Then, we explore the possibility to involve C# and .NET technologies in the journey of expanding access to spatio-temporal data managament to a broader audience represented by the .NET community. Finally, we provide and discuss the implementation of a C# library which gives access to the MobilityDB's efficient spatio-temporal data types and functions.

Chapter 1

MEOS & MobilityDB

Handling trajectory data in a database can quickly become untidy and memory intensive. Indeed, suppose storing, at small time intervals, the positions of hundreds of buses, trams and metros as single rows in a database. Of course, aggregating the rows to build a trajectory would work. However, this comes with disadvantages such as a lot of logic within the database queries as well as a messy database as the number of trajectories and positions increases. An ideal database for handling such trajectories would provide an abstraction of the logic but also represent the data in a tidy manner while reducing the data size. This is exactly the purpose of **MobilityDB**, which is covered in this first chapter.

1.1 PostgreSQL and MobilityDB

First of all, PostgreSQL is an open-source database management system written in C. PostgreSQL allows one to create extensions which brings the possibility to implement new data types, operations, indexes, functions, etc. One famous extension is "PostGIS", which brings the possibility for geographical data to be stored and queried but also indexed [14]. Since PostgreSQL has a very active community and PostGIS is a well-known extension, MobilityDB is an extension that is built on top of PostGIS and, consequently, on top of PostgreSQL [19]. The benefits of building MobilityDB as a PostgreSQL extension are many :

No need to write everything from scratch A first, intuitive, advantage is that the development can be focused on the core features. Indeed, there is no need to re-develop all the features of a DBMS[1] from scratch, such as base types (for example integers, text, floats), operations and functions (average, count, sums) but also the query optimizer, the indexing system, etc.

Taking advantage of a large and active community PostgreSQL, thanks to its vast community, has a large choice of extensions and software that allows developers to be more productive and find extensions that suits their needs. Thus, building MobilityDB as an extension to PostgreSQL lets other extensions to be used along MobilityDB.

Better integration with other technologies When creating an application involving a database, we need a connector/driver for the DBMS we are using. Developing MobilityDB as an extension for PostgreSQL avoids the creation and maintenance of such connectors for several programming languages. Indeed, PostgreSQL features connectors for many languages, either provided by the community or not.

[1] Database Management System

1.2 An Example of MobilityDB Usage

The purpose of MobilityDB is to facilitate the manipulation of **temporal** and **spatio-temporal** data [19]. Concretely, MobilityDB provides abstract data types to represent moving data and/or data that evolves over time [19]. The (spatio-)temporal types that are provided by MobilityDB are the following [19] :

- Temporal boolean
- Temporal integer
- Temporal float
- Temporal text
- Temporal geometry[2]
- Temporal geography[3]

When using MobilityDB, we can imagine several simple scenarios such as storing the evolution of humidity in a room. We can also think of more complex scenarios where we could represent the evolution of the speed of a vehicle over time as well as its geographical position which could model a trip of this vehicle. Listing 1.1 shows the creation of a table representing that scenario. Then, Listing 1.2 shows how we can insert temporal data in the table. The first row that is inserted represents the first minute of a train departing from Leuze-en-Hainaut (Belgium) on 31rd of July 2023, and the second is a very small trip on the road next to Keflavik's airport (Iceland) on January 1st 2023. Finally, Listings 1.3 and 1.1 present a query to retrieve data ordered by average speed and the associated result. As seen on the Listing 1.1, we use `tavg` (temporal average), one of the many functions (around 2300) provided by MobilityDB to perform operations on its temporal types [20].

```
CREATE TABLE Trips (
    Id int PRIMARY KEY,
    Speed tfloat,
    Position tgeogpoint
);
```

Listing 1.1: Creating a table using MobilityDB's types

```
INSERT INTO Trips (Id, Speed, Position) VALUES

  (1, tfloat '{0.0@2023-07-31 16:20:00, 83.9@2023-07-31 16:20:30, 135.12@2023-07-31 16:21:00}',
    tgeogpoint '{[POINT(3.616898 50.600578)@2023-07-31 16:20:00, POINT(3.611057 50.598917)@2023
    -07-31 16:20:30, POINT(3.596446 50.594744)@2023-07-31 16:21:00]}'),

  (2, tfloat '{18.5@2023-01-01 08:00:00, 20.0@2023-01-01 08:00:30, 85.2@2023-01-01 08:01:00}',
    tgeogpoint '{[POINT(-22.6056 63.9850)@2023-01-01 08:00:00, POINT(-22.613521 64.002337)@2023
    -01-01 08:00:30, POINT(-22.599610 64.001599)@2023-01-01 08:01:00]}');
```

Listing 1.2: Inserting temporal data into a table

```
SELECT Id, Speed, Position, tavg(speed) AS savg FROM Trips
  GROUP BY Id, Speed, Position
    ORDER BY savg;
```

Listing 1.3: Select query over temporal data

[2]Geometry represents data in a 2D cartesian plane
[3]Geography is similar to geometry except that it is used to represent coordinates in a round-earth system

	id [PK] integer	speed tfloat	position tgeogpoint	savg tfloat
1	2	{18.5@2023-01-01 08:00:00+01, 20@2023-01-01 08:00:...	{[0101000020E61000002575029A089B36C0AE47E17A14FE4F40@2023-01-0...	{18.5@2023-01-01 08:00:00+01, 20@2023-01-01 0...
2	1	{0@2023-07-31 16:20:00+02, 83.9@2023-07-31 16:20:3...	{[0101000020E610000012BEF73768EF0C403A596ABDDF4C4940@2023-07-3...	{0@2023-07-31 16:20:00+02, 83.9@2023-07-31 16:...

Figure 1.1: Result of the query of Listing 1.3

Finally, MobilityDB provides operators to perform operations such as comparisons, arithmetic operations, etc. It also provides indexes which allows for faster queries by building data structures [19]. We will not be delving more into theses topics within this work. Please note that more information can also be found within the documentation[4]. Zimányi et al.[19], the creators of MobilityDB, also provide a detailled paper about MobilityDB [19].

1.3 MobilityDB Type System

As explained in the previous section, MobilityDB provides temporal types. A temporal type is a type that represents the evolution of data within time [19]. For example, if we have a `geogpoint` that represents a position on the earth, its temporal type variant (`tgeompoint`) represents a point that moves as time goes by. A temporal type is always defined by two other types : the **base type** and the **time type** [19]. Every combination of a base type and a time type represents a **temporal type** [19].

1.3.1 The Base Type

The first part of the definition of a temporal type is its base type. Indeed, to understand what does the temporal type represent, we have to define *what* kind of data is stored. For example, if we would like to store the evolution of the altitude of an aircraft, we would like to store an integer that evolves in time. Nevertheless, if we would like to represent the evolution of the position of the same aircraft, we would like to store a geographical point which evolves in time. In order to tailor temporal types to almost all possible cases, MobilityDB supports six different base types [19]:

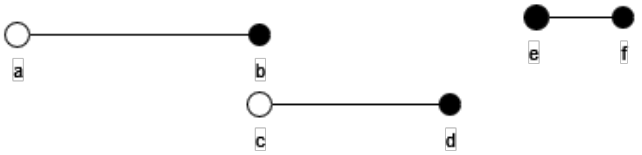
- Boolean
- Integer
- Floating number
- Text (string)
- Geometry point
- Geography point

1.3.2 The Time Type

After defining the base type of a temporal type, we also need to take in consideration the time type. Obviously, when data evolves, we need to understand *how* it evolves. For instance, suppose we would like to store accidents on a tram line. To achieve this, we need to store the (single and only) position of the tram which had an accident as well as the time at which it occurred. Another example would be the case where we want to track the trips of the same tram. In that case, we need to store the position of the tram several times and regularly. Thence, depending on the needs, the time type that will be chosen will be different. To suit all cases, MobilityDB provides four time types [19]:

[4] `https://docs.mobilitydb.com/MobilityDB/develop/mobilitydb-manual.pdf`

(a) An example of a period from a (exclusive) to b (inclusive)



(b) An example of a period set from a to f

Figure 1.2: Comparison between Period and Period set types

Timestamp Represents a single timestamp.

Timestamp set The timestamp set represents a set of "Timestamp" types. Thus, the "Timestamp" type represents a single timestamp while the "Timestamp set" represents a **non-empty** set of "Timestamp".

Period The period represents a continuous mapping between two time instants. In other words, the period represents a mathematical interval between two time instants. Let a be the lower bound timestamp and b the upper bound timestamp, we can see the period defined by a and b as $[a, b[$. Obviously, the bounds can either be inclusive or exclusive depending the use case. Figure 1.2a gives an example of a period starting in a (exclusive) and ending in b (inclusive).

Period set Naturally, a "Period set" is a set of disjoint and non-adjacent "Period" types. The advantage of the period set in comparison to the period is that the period set can, then, have gaps within its time dimension while the period can not [19]. Figure 1.2b illustrates an example of a period set.

It is clear that the `Period set` can represent the three other types. In other words, `timpestamp`, `timpestamp set` and `period` can be generalized to a period set [19]. Indeed, a period can be represented by a period set with a unique period in it. Also, a period can be reduced to a single timestamp if the lower bound of the period is the same as the upper bound. From there, we can build a set of timestamps with a set of periods. However, the idea of using period set to represent everything is not ideal. Indeed, if we specialize the time type to our needs, MobilityDB may, in some cases, have implemented more optimized algorithms for these (more specialized) types [19]!

1.3.3 Temporal Types and Notation

As explained at the beginning of this section, the temporal types are combination of a base type and a time type. The current section is summary of what is explained by Zimányi et al.[19] in their paper. However, since that paper was written, an architectural change about temporal types was implemented [12].

Since there are four time types allowing us to create temporal types, one would assume there would be four different temporal types that could be associated to their value (base type). In fact, that is not the case and we have three temporal types. These types are `Instant`, `Sequence` and `Sequences` [12]. The following subsections describe these different temporal types. For these sections, we will define β as the set of the available base types : $\beta = \{$`boolean`, `text`, `integer`, `float`, `geompoint`, `geogpoint`$\}$. Figure 1.3 shows a comparison between all these temporal types.

Instant Temporal Type

The `Instant` defines, for any base type $b \in \beta$, a single pair which associates the value of the base type b with a `timestamp` time type [19]. For instance, on Figure 1.3, the geometry point $(1, 1)$ is associated to the timestamp t_1. Thus, in the same example, we have a pair $((1, 1); t_1)$.

Sequence Temporal Type

The `Sequence` temporal type defines a period of time in which there are timestamps associated to the value of the base type. In other words, we have a set P of pairs (t, v) where t is a timestamp and v a value of the base type. `Sequence` type can either be used to represent the temporal type associated to the `Timestampset` time type or the `Period`. Indeed, the difference between those two time types is very small in the sense that `Period` includes interpolation while `Timestampset` does not define what is between two instants. The Figure 1.3 illustrates the difference between `Sequence` with and without linear interpolation.

Sequences Temporal Type

Finally, the `Sequences` type is extremely similar to the `Sequence` type. The only difference lies in the continuity of the data. Indeed, the `Sequences` is a set of `Sequence` which are non-overlapping and non-adjacent (otherwise, we could directly use a simple `Sequence`) [19]. In other words, the `Sequences` temporal type allows for "gaps" without any mapping, as shown on Figure 1.3.

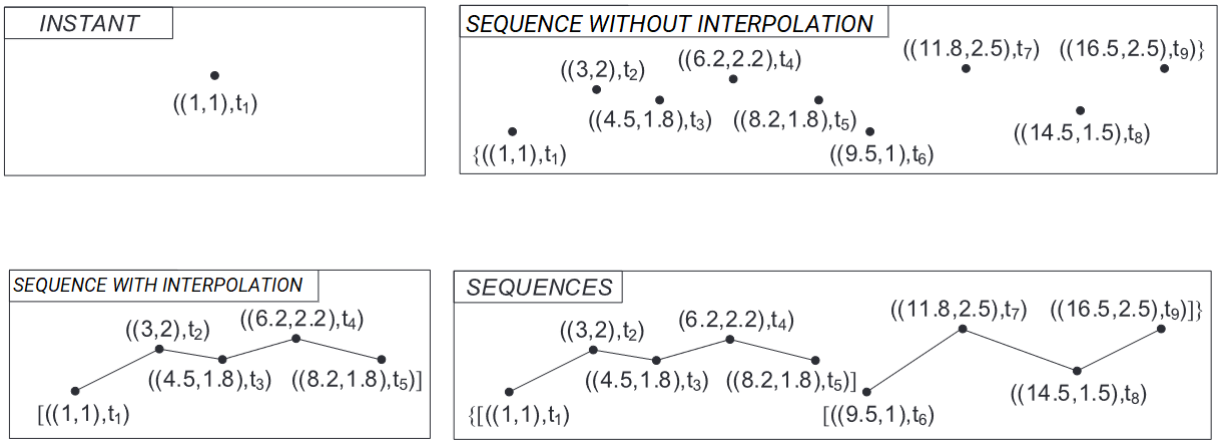


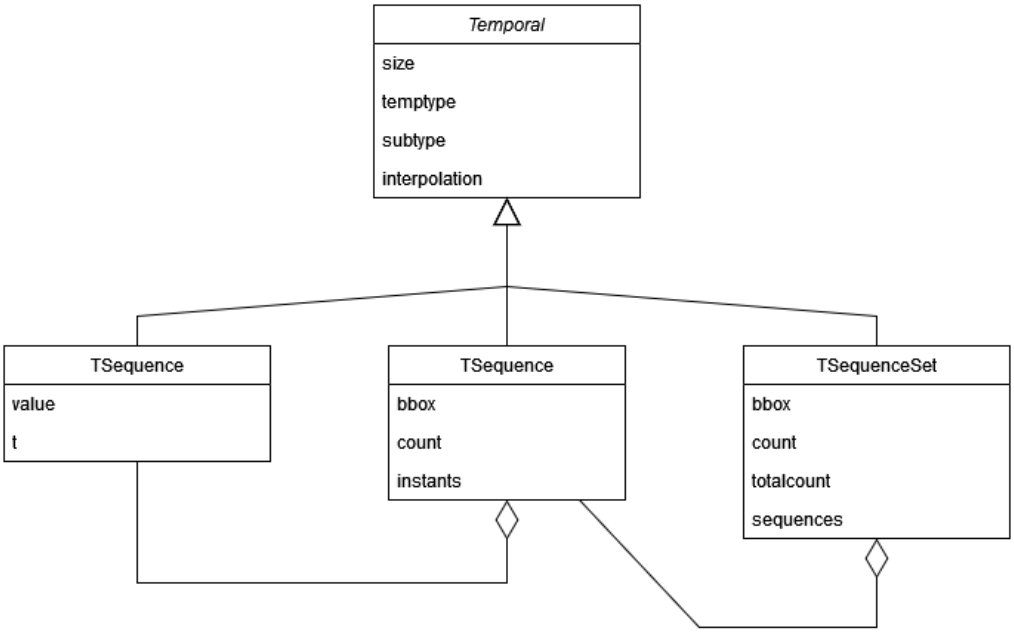
Figure 1.3: Comparison of the three different temporal types[5]

1.3.4 Implementing Temporal Types

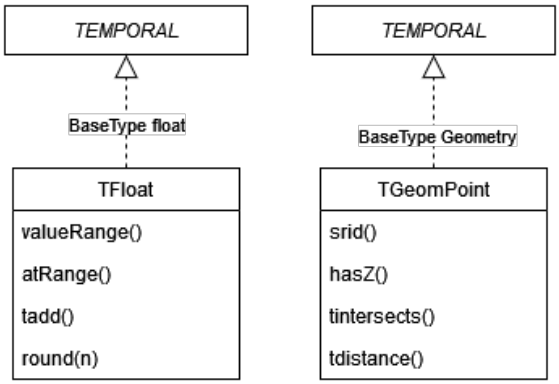
To implement the temporal types into a programming language, we use the Figure 1.4 provided by Zimányi et al.[19]. As we observe the picture, we understand that we have all three temporal types that derive from an abstract class "Temporal" which will receive the *base type* as parameter. Furthermore, the `Sequence` temporal type can have several instances of the `Instant` type, which seems obvious since they are sets of individual instants. Finally, the `Sequences` temporal type does have one or several instances of the `Sequence` simply because it is a set of that type. If we want to create an instance of a temporal type, we have to instantiate one of these classes with the base type (text, bool, integer, float, geometry or geography) as parameter.

The operations that are performed on the temporal types are **polymorphic** [19]. This means that some functions can be executed independently of the base and time type and are the same whatever the temporal type is. Some other functions only depend on the time type, and then, are

[5]Illustration strongly inspired from the paper named "MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS" by Zimányi et al. [19] as well as MEOS' documentation [12]



(a) The UML diagram of the type system and examples of temporal classes



(b) Examples of implementations of temporal classes

Figure 1.4: UML diagram of the temporal classes[6]

executed regardless of the chosen base type [19]. Obviously, these are functions that operate only on the time dimension of the temporal type. Finally, there are functions which are very specific to the base and time type [19]. For example, it is quite obvious that a function to check if two booleans intersect makes no sense while it makes complete sense for two geometry points or two geographical points. The same function would also work differently depending on the time type that is chosen. To summarize, there are various temporal types that can be instanciated, all of them having their characteristics and features.

When using a programming language that has good object-oriented capabilities, this schema is quite easy to implement. Indeed, using the concepts of **abstract** class for the "Temporal" class and **inheritance** to have the three temporal classes as children of "Temporal" is easy. Finally, we can use what is sometimes called "**genericity**" or "**template classes**" to have the base type passed in as a parameter [19].

[6]Illustrations strongly inspired from the paper named "MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS" by Zimányi et al. [19] as well as MEOS' documentation [12]

1.4 Why MobilityDB?

A question that intuitively arises would be "Why is MobilityDB needed?". Of course, we could design a database, without MobilityDB, where we store every single change of the data in a new row. Such a construction would work, but would require 1) lots of efforts and 2) complex SQL queries.

1.4.1 An Example Without MobilityDB

Let us consider an example using only PostGIS and avoiding MobilityDB. The example we show is based on M. Sakr's example (who is a MobilityDB co-founder).[7] To support the understanding and summarize the example, the paper written by J. Godfrid et al. [4] has been used.

First of all, we consider a table (shown on Listing 1.4) representing GPS points that are observed for several trips of different vehicles. The table contains a reference to the trip identifier (`tripID`) to which it is associated, a point identifier (`pointID`) to identify the observation, `t` is the timestamp at which it was observed, and, finally, `geom` represents the position of the vehicle at time `t`. Then, we create a second table (shown on Listing 1.5) representing publicity billboard spots with `billboardID` being the identifier of the billboard and `geom` its position.

```
1 CREATE TABLE GpsPoints (tripID int, pointID int, t timestamp, geom geometry(Point, 3812));
```

Listing 1.4: Creation of the GPSPoints table

```
1 CREATE TABLE Billboards (billboardID int, geom geometry(Point, 3812));
```

Listing 1.5: Creation of the Billboards table

Now, suppose we have data in those two tables and that we want to retrieve the billboards which are visible for a trip, and this, for how much time it is visible. We will consider that a billboard is visible if and only if a bus is within a close range (30 meters) to the billboard. We can create the query shown on Listing 1.6. However, the query returns the billboards that have been visible during a trip but we have no information about how long the billboard has been visible. A query that would give a good result is shown on Listing 1.7. It is clear that such a query requires one to have some serious SQL skills to be understood and written. Indeed, the query creates temporary tables, functions and other advanced concepts. To summarize, the query lists each pair of points which lies on the same trip and connects them in the right order thanks to a linear interpolation [4]. From there, we identify the points which start or end within the field of visibility of a billboard and the elapsed time is computed assuming constant speed [4].

```
1 SELECT tripID, pointID, billboardID
2 FROM GpsPoints GPS, Billboards B
3 WHERE st_dwithin(GPS.geom, B.geom, 30);
```

Listing 1.6: Retrieving which billboards which have been visible

Thus, the query on Listing 1.7, even if complex, works. However, writing such complex queries for a whole application can be long and causing bugs. Furthermore, the query is emulating a **continuous** trajectory from **discrete** GPS points stored in our table [4]. Using MobilityDB allows to make such queries much easier while working with a system that is built to work with continuous data seamlessly [4].

[7] https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/analyzing-gps-trajectories-at-scale-with-postgres-mobilitydb-amp/ba-p/1859278

```sql
WITH pointPair AS(
  SELECT tripID, pointID AS p1, t AS t1, geom AS geom1,
    lead(pointID, 1) OVER (PARTITION BY tripID ORDER BY pointID) p2,
    lead(t, 1) OVER (PARTITION BY tripID ORDER BY pointID) t2,
    lead(geom, 1) OVER (PARTITION BY tripID ORDER BY pointID) geom2
  FROM gpsPoint
), segment AS(
  SELECT tripID, p1, p2, t1, t2,
    st_makeline(geom1, geom2) geom
  FROM pointPair
  WHERE p2 IS NOT NULL
), approach AS(
  SELECT tripID, p1, p2, t1, t2, a.geom,
    st_intersection(a.geom, st_exteriorRing(st_buffer(b.geom, 30))) visibilityTogglePoint
  FROM segment a, billboard b
  WHERE st_dwithin(a.geom, b.geom, 30)
)
SELECT tripID, p1, p2, t1, t2, geom, visibilityTogglePoint,
  (st_lineLocatePoint(geom, visibilityTogglePoint) * (t2 - t1)) + t1 visibilityToggleTime
FROM approach;
```

Listing 1.7: Retrieving which billboards which have been visible (with duration)

1.4.2 Improving Queries with MobilityDB

Let us now migrate our example from a simple PostGIS implementation to a MobilityDB usage. First of all, our table to represent the points of the trips changes and a single trip will fit in a single row. Listing 1.8 shows the migration of data from the `GpsPoints` table to a new table called `Trips` using temporal geometry points, a MobilityDB type. We see that we enable MobilityDB extension and all its dependencies, and we create our trips with the data that was in our `GpsPoints` table. From there, we can express the query presented on Listing 1.9. The query nests several functions, let us deconstruct it. The nesting of `getTime`, `atValue`, `tdwithin` returns the set of periods where our position has been within 30 meters of the billboard. The `atPeriodSet` function limits a trip to the set of periods calculated just at the step before. The "WHERE" clause allows to filter the trips that have at least been at one moment within the field of visibility of a billboard.

```sql
CREATE EXTENSION MobilityDB CASCADE;

CREATE TABLE Trips (tripID, trip) AS
  SELECT tripID, tgeompointseq(array_agg(tgeompointinst(geom, t) ORDER BY t))
    FROM GpsPoints
    GROUP BY tripID;
```

Listing 1.8: Migrating the GPSPoints table into a MobilityDB Trips table

```sql
SELECT astext(atperiodset(trip, getTime(atValue(tdwithin(a.trip, b.geom, 30), TRUE))))
FROM Trips T, Billboards B
WHERE dwithin(T.trip, B.geom, 30)
```

Listing 1.9: Retrieving billboards which have been visible (with duration) using MobilityDB

From the query presented on Listing 1.9, we can conclude that MobilityDB has a natural way of processing a continuous trajectory and simplifies queries that would be complex without it.

1.4.3 Space Efficiency

We have seen in Subsections 1.4.1 and 1.4.2 that using MobilityDB reduces the complexity of the SQL queries drastically. Yet, this is not the only advantage of MobilityDB among other DBMS to manipulate trajectory data. Indeed, MobilityDB proved to be very space efficient due to its seamless way to work with a continuous path rather than a discrete sequence of points.

Using a single row Instead of creating a new row each time there is a data that evolves on a temporal type, MobilityDB stores the update in the same row thanks to the types that extend PostgreSQL. Further than bringing a database that looks more tidy, this strategy allows for other benefits that we cover hereafter.

Redundancy-removal at a constant speed on a straight MobilityDB analyzes the evolution of the data to understand if there is a real interest to keep the data or not [4]. For example, consider a vehicle moving at a constant speed, in a straight line for some time, which could be a typical case for an aircraft. Then, it is clear that we can remove the observations that have been made in between the start and the end of that straight, constant speed, line [4]. That is exactly what MobilityDB does.

Redundancy-removal when stationary Another factor of data reduction occurs when removing useless observations when they do not change. For instance, imagine a car that gets stuck in a traffic jam [4]. It is clear that the car could send several observations of its state while it did not move a tiny bit. Obviously, we can remove the observations that are in between the moment the car stopped and started to move again [4]. Suppose t_1 is the moment the car stopped and t_2 the instant it began to move again. Then, we can discard the observations between t_1 and t_2 and simply store the range $[t_1; t_2[$ associated to a zero-speed.

Clearly, MobilityDB focuses on reducing the data size of the trajectories by **compressing** them. Instead of storing every observation, MobilityDB can discard useless observations but also encode data in a smart way. Indeed, in the same way as video compression (such as MPEG) does, MobilityDB can use delta-encoding to encode its temporal types [19]. Delta encoding will not be covered within this thesis and further details can be found in the paper of Zimányi et al.[19]. However, to make it simple, delta encoding is about encoding the evolution of a given piece of data from the difference with its preceding [19]. For example, if we make an observation o_1 and an observation o_2, we can merely encode the difference between o_2 and o_1 [19]. When decoding, the delta can be reversed to retrieve the complete value [19].

A question that arises now that we are aware of the concepts used by MobilityDB to reduce data size. Indeed, we could question ourselves whether those techniques are really that efficient in practice. The answer is yes. Several examples of real-data migration to MobilityDB resulted in data compression with stunning compression rates.

Moscow's public transportation Experiments have been made on the Russian's capital transportation system [4]. For a period of a single day, which represented 500 MB of data (10 billion rows), MobilityDB could reduce that to 5 MB (15000 rows) [4]. This is a compression ratio of 100:1!

Aircraft flights data In their paper, A. Broniewski [1] exposed the queries needed to reshape public aviation data[8] into a MobilityDB temporal database. Avoiding details, they could obtain a database of 870 MB with MobilityDB while the base dataset had the size of 2531 MB, and this, for a volume of data which represents a single day of flights [1]. In other words, they could make their database $2531/870 = 2.91$ times smaller than the original for a period of 24 hours [1].

[8] Public dataset available from `https://opensky-network.org/data/datasets`

1.4.4 Faster Queries

Finally, MobilityDB proposes queries which result in smaller execution times. One example, presented by Zimányi et al.[20], is based on a BerlinMOD, a benchmark for spatio-temporal DBMS, developed by the Fern University of Hagen.[9] [3] The same university did develop an extensible spatio-temporal database system named "SECONDO".[10] When comparing MobilityDB and SECONDO using 12 queries with different database sizes, it came out that, for 63% of the queries, MobilityDB was faster [20]. A stunning result is that the total time to run all the queries was only 13% of the time needed by SECONDO [20].

1.5 MEOS and MobilityDB Architecture

As described in Section 1.3, MobilityDB provides temporal types. At the early stages of the project, the temporal types were not isolated in a dedicated component of the project. As the time evolved and the project growing up, with the desire to integrate MobilityDB with other technologies, the developers extracted the manipulation of temporal types and spatio-temporal data in a dedicated component. That dedicated component is called "MEOS" (acronym for "Mobility Engine, Open Source") [12]. Obviously, MEOS is a dependency of MobilityDB [12]. However, this means that MEOS can be used alone for other projects [12]. Such a library suggests that one would be able to create other projects based on MEOS. For example, wrappers for MEOS can be created in other programming languages, such as PyMEOS.[11] From there, we could imagine a scenario where we would be able to query a database, retrieve the temporal and spatio-temporal types, and play with them within a project using another technology.

[9] `https://www.fernuni-hagen.de/`
[10] `https://secondo-database.github.io/`
[11] A Python wrapper for MEOS : `https://github.com/MobilityDB/PyMEOS`

Chapter 2

An Introduction to Microsoft .NET Platform

.NET is a Microsoft platform which provides a unified environment to build applications for several platforms. Over the years, Microsoft improved .NET up to a reference platform for developers which is extremely versatile and popular.

2.1 A Quick Introduction

In order to provide code interoperability and reusability, Microsoft introduced, in 2002, what is called .NET (pronounced and also called "dotnet") nowadays. The goal of Microsoft was to provide an efficient, versatile platform to build different kinds of applications such as desktop, mobile, web applications but also games, etc [11].

After many years of development, .NET became a strong **open-source**, **unified** framework supported by many platforms. Companies and developers can use .NET to write applications with one single API and with several programming languages. With more than two decades of existence, .NET has become very mature and the community as well as Microsoft themselves contribute to the creation of new libraries extending the functionality of the framework making it rich in features [11].

2.2 Tracing the History of .NET

From the initial release of what was called ".NET framework" in 2002 to the most recent and mature versions of .NET, Microsoft went through a long journey.

2.2.1 A Good Start with .NET Framework

At the early stages of .NET, Microsoft released several versions under the name ".NET framework", which was only available under Windows [13]. The first release proposed tools to build desktop applications under Windows as well as web applications. Since 2002 and up to 2022, Microsoft updated .NET framework with new features until version 4.8. Newer versions of .NET framework provide new types of applications such as games, mobile applications, services, web workers. From 2022, Microsoft continues to service .NET Framework's latest releases with security and reliability fixes but no new features due to the transition to .NET Core [10] which begins to be very mature. Table 2.1 shows a list of the most important .NET framework versions that have been released.

.NET Framework version	Release date	End of support
.NET Framework 1.0	15 January 2002	?
.NET Framework 1.1	9 April 2003	?
.NET Framework 2.0	17 February 2006	12 July 2011
.NET Framework 3.0	21 November 2006	12 July 2011
.NET Framework 4.0	12 April 2010	12 January 2016
.NET Framework 4.5	9 October 2012	12 January 2016
.NET Framework 4.5.1	15 January 2014	12 January 2016
.NET Framework 4.5.2	5 May 2014	12 January 2016
.NET Framework 4.6	29 July 2015	26 April 2022
.NET Framework 4.6.1	30 November 2015	26 April 2022
.NET Framework 4.6.2	2 August 2016	12 January 2027
.NET Framework 4.7	11 April 2017	-
.NET Framework 4.8	18 April 2019	-
.NET Framework 4.8.1	9 August 2022	-

Table 2.1: Important releases of .NET Framework [8]

2.2.2 Evolution to .NET Core

With desire to have a more versatile framework, Microsoft decided to re-write .NET framework completely and start over again with .NET Core which brings the most massive changes since the introduction of .NET framework [13]. Reworking the code to work with **more modern** design patterns, going **open-source** and an aim for a **cross-platform** framework [13], Microsoft released .NET Core 1.0 on 21$^{\text{th}}$ June 2016 [7]. Table 2.2 shows the important versions of .NET Core that were released since 2016.

.NET Core version	Release date	End of support
.NET Core 1.0	27 June 2016	27 June 2019
.NET Core 1.1	16 November 2016	27 June 2019
.NET Core 2.0	14 August 2017	1 October 2018
.NET Core 2.1	30 May 2018	21 August 2021
.NET Core 2.2	4 December 2018	23 December 2019
.NET Core 3.0	23 September 2019	3 March 2020
.NET Core 3.1	3 December 2019	13 December 2022

Table 2.2: Important releases of .NET Core [7]

2.2.3 Recent Versions of .NET

Since 2022, Microsoft does not release new versions of .NET framework (see Section 2.2.1) but continues the development of .NET Core, which only has advantages. After .NET Core 3.1, Microsoft released ".NET 5", removed "Core" from the name and skipped version 4 to point out that new projects should use this and not the old .NET framework which is not supplied with new features anymore. The naming convention remains the same nowadays with newer versions of .NET. Thus, ".NET" is the continuity of the ".NET Core" project while ".NET framework" is a complete different (and old) framework. Table 2.3 shows the recent versions of .NET and the latest one being .NET 7. In the table, we note that .NET 5 has a smaller support length in comparison to .NET 6. Indeed, Microsoft decided that every odd-numbered version of .NET will be a classic support length (18 months) while the even-numbered versions will benefit from a long-term support (LTS) of 3 years [17].

.NET version	Release date	End of support
.NET 5.0	10 November 2020	10 May 2022
.NET 6.0	9 November 2021	12 November 2024
.NET 7.0	8 November 2022	14 May 2024

Table 2.3: Latest releases of .NET [7]

The idea of Microsoft is to encourage developers to make the transition to .NET when possible. It is, of course, easy when your application is running under .NET Core since it is a simple version update which is a simple parameter to be changed in your .NET application [17]. However, it is not that easy when dealing with an application that was built using .NET framework since the codebase is not the same. Furthermore, in some cases, you may want to stay with .NET framework for your application for compatibility issues and/or libraries you would be using. In that case, staying with .NET framework is still good and Microsoft still provides security and reliability patches for .NET framework. Anyhow, for new developments, it is strongly advised to go with the latest version of .NET.

2.3 Cross-Platform Capabilities

A major interest for recent .NET versions is its cross-platform capabilities. From a Windows-only support for .NET framework, Microsoft extended the support for .NET to several Linux distributions, MacOS but also different mobile platforms. Table 2.4 lists the Windows operating systems supported by .NET 7, Table 2.5 the Unix-based OS that are supported and Table 2.6 the mobile systems that .NET 7 supports [9]. Note that older operating systems and/or older versions of them may be supported in older versions of .NET while they are not for .NET 7 [9].

OS	Version
Windows 10	Version 1607 and after
Windows 11	Version 22000 and after
Windows Server	Version 2012 and after
Windows Server Core	Version 2012 and after
Windows Nano Server	Version 1809 and after

Table 2.4: List of Windows OS supported by .NET 7 [9]

OS	Version
Alpine Linux	Version 3.15 or higher
CentOS Linux	Version 7
CentOS Stream Linux	Version 10
Debian	Version 10 or higher
Fedora	Version 36 or higher
openSUSE	Version 15 or higher
Oracle Linux	Version 7 or higher
Red Hat Enterprise Linux	Version 7 or higher
SUSE Enterprise Linux	Version 12 SP2 or higher
Ubuntu	Version 18.04 or higher

Table 2.5: List of Unix-based OS supported by .NET 7 [9]

OS	Version
Android	API 21 or higher
iOS	Version 10 or higher

Table 2.6: List of mobile OS supported by .NET 7 [9]

2.4 Language Versatility

Using .NET to design programming languages is the main strength of Microsoft. Indeed, several languages are based on .NET such as C# (which is the most famous) but also F# or VB.NET. Every one of these languages is compiled into a common intermediate language called "MSIL"(Microsoft Intermediate Language), which is then compiled into native code by the JIT (Just-in-time) compiler [13]. Such a way to work allows for language interoperability between .NET languages. For example, it is possible to write a class library in VB.NET and use the VB.NET code from another C# project, as shown on the code snippets on Listings 2.1 and 2.2 as well as the code output on Listing 2.3. However, it is impossible to mix languages within the same assembly.

```
1 Public Class Test
2
3     Public Sub Test(ByVal v As String)
4         Console.WriteLine("Hello, this is VB.NET called from " + v)
5     End Sub
6
7 End Class
```

Listing 2.1: A class library written in VB.NET

```
1 using VBCode;
2
3 var c = new Test();
4 c.Test("C#");
```

Listing 2.2: A C# code using the VB.NET library of Listing 2.1

```
1 Terminal Output:
2 Hello, this is VB.NET called from C#
```

Listing 2.3: Terminal Output

2.5 The Main Components of .NET

To be able to provide cross-platform capabilities as well as support for different languages as explained in Sections 2.3 and 2.4 respectively, Microsoft had to set up a set of rules called **Common Language Infrastructure** (CLI) [18]. The CLI states that every .NET implementation should feature the **Base Class Library** (BCL), a rich set of libraries which are standard among all these implementations [18]. In this section, we cover the CLI, its key elements as well as the BCL and what it implies.

2.5.1 The Common Language Infrastructure and Common Language Runtime

The **Common Language Infrastructure**, or CLI, is a platform-independent specification designed by Microsoft which brings rules for a runtime environment and code execution with several programming languages [18]. The specification is divided in three parts :

Common Type System (CTS) Establishes a set of data types, their representation in memory and rules to ensure the compatibility of these types across the different languages supporting the Common Type System [18]. Note that the restrictions which are set to ensure language interoperability are also sometimes called **Common Language Specification**

Standardized metadata Defining standard metadata to describe the structure of a program is the purpose of this component [18].

Virtual Execution System Defines the runtime environment namely the type instantiation, destruction, but also how the different data types will interact [18]. One of the main definitions that it gives is the common, intermediate, language described in Section 2.4.

The CLI specification is implemented by Microsoft as the **Common Language Runtime** (CLR) [18]. Thus, the CLR is what gives the developer runtime services and capabilities such as :

- Type safety
- Memory management using garbage collection mechanism
- Just-in-time compilation and/or ahead-of-time compilation
- Exception handling
- Thread management
- Reflection

2.5.2 The Base Class Library

In Section 2.5.1, we described the CLI (Common Language Infrastructure) as the specification describing the runtime environment on which the code following the specification will run. Further than providing the services to execute code with the CLI implementation, called CLR (Common Language Runtime), .NET developers wanted to provide a set of standard libraries and APIs [18].[1] With the aim of doing this, Microsoft introduced the **Base Class Library**.

The BCL is a rich set of pre-built classes and APIs that a developer can use. It defines complex types (objects) which allows the developer to implement common operations in an easy manner. The BCL can be used from any language implementing .NET.[2] For example, the BCL gives the ability to manage collections such as lists, queues, stacks, etc. Listing all the features that the BCL provides is almost impossible due to the vast amount of capabilities it offers, but here is a small list of possibilities that .NET's Base Class Library proposes :

- File reading and writing, streams handling
- Thread management
- Network communication (TCP, UDP, HTTP, sockets and other network protocols)
- Collections management (Queues, stacks, lists, dictionaries and so on)
- String manipulation
- XML manipulation

[1] API : Application Programming Interface
[2] In this case, we are talking about C#.NET, F#.NET and VB.NET

- JSON manipulation
- Data querying and manipulation
- Reflection
- Interaction with processes and system
- Security implementation such as cryptographic functions, authentication, etc.

The combination of the CLR and the BCL together allow a developer, using any language following the CLI specification (what we call a ".NET language") to use a rich set of APIs but also to benefit from the language interoperability that it offers. Listing 2.4 is an example of a C#.NET code retrieving an XML file containing books, changing the title of every book node, and saving it to the same file.

```
using System;
using System.Xml;

XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("data.xml");

XmlNodeList bookNodes = xmlDoc.SelectNodes("//book");

foreach (XmlNode bookNode in bookNodes)
{
    XmlNode titleNode = bookNode.SelectSingleNode("title");
    titleNode.InnerText = "New Title";
}

xmlDoc.Save("data.xml");
Console.WriteLine("XML manipulation completed.");
```

Listing 2.4: An example of XML manipulation code using C#.NET

2.6 Global Adoption of .NET

Over the years, .NET could conquer the market and become widely-used among companies and developers all over the world. To have an idea of the proportion of professional developers using .NET, we use the 2023 edition of the very popular developer survey of StackOverflow. The survey reports that 27.11% of the 52046 professional developers that responded are using .NET (version 5 or above) [15]. Furthermore, among 67053 professional developers, 29.16% are using C#.NET programming language, 3.92% use VB.NET and 1.03% use F# [15]. The survey also states that 62.87% of the people currently using C#.NET would like to continue with the technology in the future [15]. Finally, .NET is the most popular framework within the list of the survey [15].

It is clear that basing the results on a single survey is not the most reliable and exact source of data. However, it gives a good idea on the popularity of .NET over the world.

2.7 Advantages of .NET

A question that arises when discovering a new technology would be "Why should we use that technology above another?". So, why would .NET be a good choice among all the technologies that are available?

Wide community .NET has a large active community which allows .NET to be more open to newcomers. Furthermore, the community is very active in package development, projects, books, code sharing, etc.

Packages availability and management Due to its large community, .NET has a myriad of packages that are available. Through its package manager, called "NuGet", the packages are installed with a simple click or via a simple command.

.NET is open-source Since the release of .NET Core (see Section 2.2), Microsoft went open-source, which allowed the community to enhance .NET features and be more accessible to the developers.

Performance Since .NET languages are compiled into an intermediate language (see Section 2.4) and then to machine code, their performance is good in comparison to interpreted languages. A comparison with other compiled languages is not covered within this thesis.

Large field of possibilities Another advantage of .NET would be the amount of possibilities that it brings. Indeed, .NET allows to create desktop applications with WinForms, WPF or MAUI but also to create web applications with ASP.NET Core or even build mobile applications with Xamarin or MAUI. Furthermore, .NET can be used to create web workers, Windows services, and even games with game engines such as Unity [17].

Cross-platform capabilities As explained in Section 2.3, .NET applications can be run on a collection of different platforms allowing to have a single codebase for every platform on which an application is deployed.

Language interoperability and versatility As described in Section 2.4, .NET has been used to design several programming languages such as C#.NET, F#.NET and VB.NET. Such a design enables a developer to use any .NET language code that has been compiled, from another project written in another .NET language.

2.8 The Future of .NET

Open-source also means that the vision for the future as well as the planned features for the next releases of .NET are publicly available. Dotnet developers expose the upcoming features on a web application named "Themes of .NET" [17].

Chapter 3

Database Querying in .NET

In the previous chapters, the management of spatio-temporal data with MobilityDB and MEOS as well as the .NET ecosystem have been covered. Before trying to merge those two concepts into a practical work, we may want to understand how to query a database with .NET. Indeed, the practical work presented within this thesis is going to require the knowledge of database querying with .NET technologies. This chapter will quickly cover basic knowledge of how to query databases in .NET using *Entity Framework Core* (also called "EF Core" or "EF"). For comprehensive and in-depth explanations about EF Core, the book "Practical Entity Framework" by Brian L.Gorman [5] provides valuable insights and extensive analysis.

3.1 Older Techniques

In the past, older technologies and techniques existed to query databases. Obviously, the approach to query databases has drastically changed to steer clear of vulnerabilities but also to be more efficient and convenient. Older technologies that were the most famous among the .NET community are *OLEDb* (Object Linking and Embedding Database Object) and *ADO.NET* [5]. However, in this work, we will only concentrate our efforts on a more modern way to query databases in .NET.

3.2 Object-Relational Mapping

Object-relational mapping is a more modern technique to achieve database querying. Indeed, instead of creating the type manually each time after a query, the developer creates a class reflecting the properties of the target table. From there, what we call an **ORM** (object-relational mapper) will understand the model class and how to map from the table to such a class. For example, the table presented in Table 3.1 will be mapped by an ORM to the C# class presented in the Listing 3.1. Note the decorators above the C# properties to translate the lowercase column names of PostgreSQL to uppercase property names of C#.

Column	Type	Remark
id	int	Primary key
name	varchar(50)	
birthdate	date	

Table 3.1: A PostgreSQL table representing users

```
class User
{
    [Key]
    [Column("id")]
    public int Id { get; set; }

    [Column("name")]
    public string Name { get; set; }

    [Column("birthdate")]
    public DateTime BirthDate { get; set; }
}
```

Listing 3.1: The users table of Listing 3.1 mapped to a C# class by an ORM

3.3 Entity Framework Core

As explained in the previous section, we will make use of an ORM. There exists myriads of object-relational mappers for .NET, such as *Dapper*[1] or *NPoco*.[2] The main focus during this work will be on *Entity Framework Core*, which was first introduced in 2008 [5]. Entity Framework Core is owned and maintained by Microsoft continuously. Also, Entity Framework benefits from a solid documentation[3] and its popularity makes it easy to find online resources about it. Later in this thesis, we cover the technical reasons which led to the choice of EF Core among other ORMs.

3.4 Linq

"Language INtegrated Query" or simply "Linq" is a component of the .NET ecosystem allowing to access different sources of data using the same syntax [13]. In fact, Linq is a layer of abstraction that is added to access data. Starting with a basic collection to more complex applications such as XML or even SQL, Linq provides us with a unique syntax. The main advantage of that, is that if we come to a change of data source, the changes are minimal. At the end of the day, Linq provides two different syntaxes to query data sources. The Listing 3.2 shows the first type of syntax which has similarities with SQL, while the Listing 3.3 shows the second way of proceeding, which is based on **extension methods** [13].[4] Both codes perform the same operation, namely getting the first 3 characters of the strings which contains "hey". With the aim of standardization, we will use the second syntax (extension methods-based) for the rest of this work.

```
using System.Linq;

// Suppose this list contains data
var strings = new List<string>();
var filteredStrings = from s in strings
                      where s.Contains("hey")
                      select s.Take(3);
```

Listing 3.2: A Linq query on a collection using the "SQL-like" syntax

[1] https://github.com/DapperLib/Dapper
[2] https://github.com/schotime/NPoco
[3] https://learn.microsoft.com/en-us/ef/core/
[4] Extension methods are static methods that can "extend" the features of a given type https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

```
using System.Linq;

var strings = new List<string>();
var filteredStrings = strings
    .Where(s => s.Contains("hey"))
    .Select(s => s.Take(3));
```

Listing 3.3: A Linq query on a collection using the extension methods-based syntax

To summarize the backgrounds of Linq, it provides *interfaces* and *methods* which are then implemented by *providers* [13]. Clearly, if one creates a class that implements those interfaces and methods, he could create a *provider* for a new data source. For instance, suppose we decide to build a new file format, we could create a Linq provider to query it with the standardized Linq syntax! For further information regarding Linq, C. Nagel provides quality material in his book "Professional C# and .NET" [13]. Additionally, the Microsoft .NET documentation is very thorough.[5]

Currently, there exists several well-known providers for diverse data sources such as XML files, Amazon DynamoDB, SQLite or even for Entity Framework.[6] The latter will be covered during the rest of this chapter.

3.5 Entity Framework Core and Linq to Entities

Now that we know what Entity Framework is, as well as Linq and the concept of Linq provider, it is time to put all of that together. Indeed, Linq possesses a provider, embedded in the .NET ecosystem, which enables the retrieval of data from Entity Framework Core. This driver is called *Linq to Entities*. The name comes from the fact that an *Entity* is the name given to a class mirrorring the content of a database table in EF Core. In the next section, a complete, simple example of a query to the database using Entity Framework Core and the Linq syntax will be presented.

3.6 A Complete Example

Now that have an insight on Entity Framework and Linq as well as the .NET ecosystem, we will analyze an elementary example of a query a PostgreSQL database in C#.NET.

3.6.1 Setting up the C# Application

For the purposes of this application, we create a simple console application where we add the dependencies for Entity Framework Core and NpgSQL in their latest stable versions. As explained above, EF Core enables type mapping to our C# models. NpgSQL is a provider for Entity Framework Core which allows to interact with PostgreSQL databases. The packages can be installed from "Nuget"[7], the .NET package manager provided by Microsoft.

3.6.2 The Database Schema

For this example, we will reuse the simple schema of the Table 3.1.

3.6.3 Creating the .NET Model

Our goal is to map the table containing the users to a C# object. In order to achieve that, we have to create the class that will contain the data. Therefore, we create properties with the

[5] https://learn.microsoft.com/en-us/dotnet/csharp/linq/
[6] Note that this list is non-exhaustive
[7] https://www.nuget.org/

corresponding types and add the annotations on top to indicate the primary key and the name of the field in the database. Listing 3.4 is the resulting class.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace EFQuerying
{
    internal class User
    {
        [Key]
        [Column("id")]
        public int Id { get; set; }

        [Column("name")]
        public string Name { get; set; }

        [Column("birthdate")]
        public DateTime BirthDate { get; set; }
    }
}
```

Listing 3.4: Creation of the model corresponding to the "users" database table

3.6.4 Setting up the DBContext

To interact with databases in Entity Framework Core, we need to create what we call a "DbContext" [5]. To do so, we create a class which derives from EFCore's "DbContext" base class, as shown on Listing 3.5. We clearly observe that we can override the "OnConfiguring" method to tell Entity Framework how to connect to our PostgreSQL database with our connection string.[8] We specify to the "OnModelCreating" method that our table is written in lowercase in the database. And, finally, we add a public property of type "DbSet" with the type of the model defined in Section 3.6.3 as parameter. Now, the `DbSet` object instance will be the type that will enable querying to the database.

```
using Microsoft.EntityFrameworkCore;

namespace EFQuerying
{
    internal class PostgreDbContext : DbContext
    {
        public DbSet<User> Users { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseNpgsql(
                "Server=localhost;Port=5432;Database=postgres;User Id=postgres;Password=admin;"
            );
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<User>()
                .ToTable("users");
        }
    }
}
```

Listing 3.5: Creation of Entity Framework's DbContext

[8] Note that the connection string should never be put in plain text in real projects

3.6.5 Querying the Database

Now that Entity Framework is set up to interact with our database schema, we can query the database. EF Core requires an instance of the `DbContext`. From the instance of the `DbContext`, we can access the Users DbSet and simply use a Linq query to retrieve data. We finally display the data with the formatting we desire. The Listing 3.6 illustrates the code used for the query and Listing 3.7 gives represents what is displayed on the standard output.[9] Note that we need to dispose the resources of the DbContext after usage, which explains the usage of the "using" block. Basically, the query asks Entity Framework (via Linq) to query the database to retrieve the users ordered by their name.

```
using EFQuerying;

using (var dbContext = new PostgreDbContext())
{
    List<User> users = dbContext.Users
        .OrderBy(u => u.Name)
        .ToList();

    foreach (var user in users)
    {
        Console.WriteLine($"User ID: {user.Id}, Name: {user.Name}, Birth Date: {user.BirthDate.
    ToString("dd/MM/yyyy")}");
    }
}
```

Listing 3.6: Creation of Entity Framework's DbContext

```
Terminal Output:
User ID: 3, Name: Annie, Birth Date: 26/02/1991
User ID: 2, Name: Jean, Birth Date: 05/09/1963
User ID: 4, Name: Paul, Birth Date: 12/04/2002
User ID: 1, Name: Thomas, Birth Date: 06/12/1999
```

Listing 3.7: Output of the code of Listing 3.6

[9] With a few rows inserted beforehand

Chapter 4

Interest for a MEOS Wrapper in .NET

It is time to delve deeper in the practice and understand the goal of this work. This chapter highlights the significance of the work behind "MEOS.NET", a .NET wrapper for MEOS and the objectives of the project. We review and provide how this project of integration of spatiotemporal capabilities into the .NET ecosystem can address notable challenges and open up new horizons in temporal and geospatial data handling in modern applications.

4.1 Objectives of MEOS.NET

As explained above, the main goal of this work is to develop a .NET wrapper for MEOS. Other wrappers for MEOS exist for other technologies such as PyMEOS for Python[1] or even JMEOS for Java.[2] Since each technology has its own specifications, the following describes the milestones of the project.

4.1.1 Exposing MEOS Functions

MEOS provides functions to manipulate spatio-temporal data. The first target is to expose these functions as a .NET class. Obviously, since MEOS is not a .NET library but a C-based library, a few things change. Indeed, the C-based library runs as **unmanaged** code while .NET is a **managed** environment. As discussed in Chapter 2, the .NET has its execution controlled by the CLR.[3] This means that .NET benefits from mechanisms such as garbage collection, type safety, exception handling. C language does not have a managed context, and thus, we need a way to communicate between our .NET **managed** code and the original MEOS code which is **unmanaged**. This process is called marshalling in .NET [6]. Fortunately, Microsoft provides tools within its .NET environment to handle such interoperability scenarios with native code [6].

4.1.2 Implementing Types with Dotnet Style

After exposing the MEOS functions in .NET, the types have to be created and the functions called appropriately. Indeed, further than simply providing the MEOS functions, the goal is to have a library which allows to manipulate plain .NET classes and object instances, with the functions encapsulated as methods of these classes/objects. In other words, the library has to be **transparent** for a .NET developer who would like to use the library. In fact, the library should handle all the marshalling and C functions calls and simply expose .NET classes that encapsulates those mechanisms, and which, of course, respects the conventions of .NET. Listing 4.1 illustrates an example of type marshalling from unmanaged code. However, letting users of the library doing that themselves is, firstly, not clean, and secondly, error-prone and could lead to unexpected behaviours. To avoid such a scenario, the Listing 4.2 is an example of how it would be possible to wrap the marshalling within a class. Finally, Listing 4.3 shows how to use the wrapped marshalling of Listing 4.1 as a "black-box".

[1] `https://github.com/MobilityDB/PyMEOS`
[2] `https://github.com/nmareghn/MobilityDB-JMEOS`
[3] Common Language Runtime

```
using System.Runtime.InteropServices;

var pointer = GetSomeDataFromUnmanagedCode();
var data = Marshal.PtrToStructure<DotnetMappedClass>(pointer);
```

Listing 4.1: An example of type marshalling using IntPtr

```
public class DotnetMappedClass
{
    private IntPtr _ptr;

    public DotnetMappedClass()
    {
        this._ptr = GetSomeDataFromUnmanagedCode();
    }

    public void ExecuteSomeFunctionOnUnmanagedCodeTransparently()
    {
        this._ptr.ExecuteSomething();
    }
}
```

Listing 4.2: Wrapping marshalling within the .NET type

```
DotnetMappedClass data = new DotnetMappedClass();
data.ExecuteSomeFunctionOnUnmanagedCodeTransparently();
```

Listing 4.3: Using type marshalling transparently

Obviously, the target for this work is to abstract the logic of the interoperability between the managed and unmanaged code for the end, MEOS.NET, user.

4.1.3 Database Driver and ORM

Finally, the ultimate goal is to be able to query a MobilityDB database plain object instances, and this, automatically. In modern technologies, such libraries and frameworks that are specialized in mapping relational tables to plain objects exist. These are called **object-relational mappers** (ORM). Examples of well-known ORM's among the community of .NET developers are *NPoco*[4], *Dapper*[5], *Entity Framework*[6], the latter being owned and maintained by Microsoft and fully part of the .NET ecosystem. However, when using an ORM, the type conversions should be declared before usage. With the aim to provide **transparency** to the .NET developers, another goal of this work would be to provide the MEOS type conversions for **Entity Framework**.

4.2 The Advantages of a .NET Wrapper

The advantages to the implementation in .NET of a MEOS wrapper are many.

Availability of efficient spatio-temporal data management in .NET A first, intuitive, advantage of having a .NET wrapper for MEOS is to bring an efficient way of working with spatio-temporal data in .NET, which is fairly hard to find nowadays.

Seamless integration With MEOS.NET, developers would easily and seamlessly manage spatio-temporal data within their applications. This could be particularly beneficial for projects dealing with location-based services, tracking applications, geospatial analysis, and transportation systems.

[4]https://github.com/schotime/NPoco
[5]https://github.com/DapperLib/Dapper
[6]https://learn.microsoft.com/en-us/ef/

Community support As discussed in Chapter 2, .NET has a vast active community. This means that bringing spatio-temporal capabilities into the .NET ecosystem could bring new enthusiasts to the domain.

Integration in modern applications .NET is an ecosystem of choice for modern applications, particularly for web applications and API's [15]. This means that providing an easy way to interact with MobilityDB in .NET would probably increase the presence of applications and projects based on MobilityDB.

Chapter 5

Proof of Concept

To demonstrate that the decision to invest in this project is warranted, a proof of concept was created. The idea behind the proof of concept is to prove that we can establish a "tunnel" between MEOS and .NET. From there, we want to show that the "tunnel" wrapping the MEOS functions is operational. A concise demonstration application has been built to prove it but we will also explain the concepts behind the scenes. Furthermore, we demonstrate how to ensure the quality of our library by implementing unit tests. Of course, the architectural and practical decisions will be reviewed. For more information about the complete code of MEOS.NET, please refer to Annex "MEOS.NET Repository".

5.1 Why C#?

As explained in Chapter 2, .NET is an ecosystem having several languages which are interoperable. This means that we could choose any .NET language, such as C# but also F# or even VB.NET. First of all, we will avoid F# since it is a functional language. Indeed, for this project, we will choose an object-oriented language since we need classes and encapsulation, but above all, inheritance, polymorphism and genericity to implement the temporal types as accurately as possible, as demonstrated in Section 1.3.4. Finally, C# has been chosen above VB.NET due to its popularity, the amount of documentation available but also its modern and clean syntax. As explained in Section 2.6, C# is among the most popular languages for professional developers and companies. C# is for sure, the most versatile, elegant and modern languages of the .NET ecosystem. This means that support from the community is more likely to happen with a C#-based library rather than with other .NET languages.

5.2 Type Marshalling in C#

In Chapter 4, we have seen that C# integrates mechanisms for **type marshalling**. It is clear that one could ask the question whether it is really important to have type marshalling in C#. Indeed, C# allows the usage of pointers in an unsafe context, as presented on Listing 5.1. However, pointers are not recommended and are avoided in almost every case [2]. The main reason for this, is that C# runs in a **managed** context, with type safety, error handling, and garbage collection. Obviously, code readability would be impacted with the usage of pointers. Using pointers would, then, go against the core philosophy of managed languages. Anyhow, if pointers are really required, the "*/unsafe*" flag needs to be set at compile time.

```
using System;

class Program
{
    static unsafe void Main()
    {
        int number = 42;
        int* pointer = &number;

        Console.WriteLine("Value: " + *pointer);
    }
}
```

Listing 5.1: Example of pointer manipulation in C# within an unsafe context

To bring safety while enabling the management of pointers in .NET, an important structure is`IntPtr`. This structure simply represents the address of a pointer without knowing the real type behind it. The advantages of using `IntPtr` is that it is a safe datatype which allows error handling with exceptions in the case an illegal operation is performed on the underlying pointer.The drawback is that, since the underlying pointer type is unknown, it could be easy to make mistakes.

In the case of MEOS.NET, the goal is simply to wrap the C functions in C# classes and pass the pointers to the right function. No MEOS function will be exposed to the end user. This means that, for MEOS.NET, we do not require more complex marshalling mechanisms and will wrap the `IntPtr` into classes that will be exposed.

5.3 Wrapping MEOS Functions as POCO Methods

Now that we have an insight on the tools and the architecture that is going to be used, we can begin to discuss *how* the things are done. In this section, we review the methodology that we will follow during the development of MEOS.NET to wrap the functions and make them available outside of the library. The final objective is to expose "Plain Old C# Objects" (POCO) with methods that can be called and that use MEOS as a blackbox.

5.3.1 Automating MEOS Functions Generation

The first step of our development is to be able to call the C functions within our C# code. To do so, the .NET ecosystem provides a tool in the `System.Runtime.InteropServices` namespace, as shown on Listing 5.2. Note that we need the C shared library and reference it as annotation above the function. Then, the function declaration is the same as the C declaration except that we will make it public to the other classes in our MEOS.NET namespace.

```
using System.Runtime.InteropServices;

[DllImport("Some.dll", CallingConvention = CallingConvention.Cdecl)]
public static extern void some_c_function(string param);
```

Listing 5.2: Calling a C function in C#

However, in our case, we have a myriad of functions to declare. Within the project, we needed a powerful, compact, reliable and highly-available script to generate the C# declarations for us. The choice of Python for such a script has been made for its simplicity to work with regular expressions, which allows us to analyze MEOS' header file, analyze the functions' signatures and output a C# signature. The Python script, called the "builder" lives under the namespace "MEOS.NET.API.Builder" and is responsible for the generation of the C# declarations that will live in the class "MEOSFunctions" of the namespace "MEOS.NET.API.Internal" along with the

shared library (see Figure 5.1). Finally, the MEOS.NET project is setup to run the functions generation each time the library has to be built, as shown on Listing 5.3.

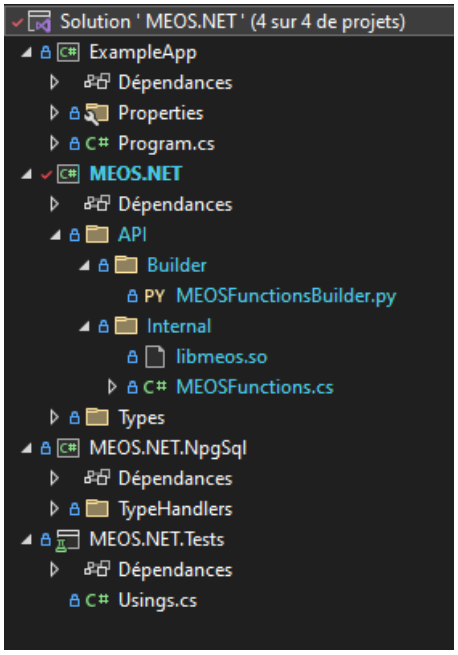


Figure 5.1: Project folders

```
<Target Name="PreBuildTask" BeforeTargets="Build">
    <Exec Command="echo Building MEOS Functions..." />
    <Exec Command="python3 $(ProjectDir)API/Builder/MEOSFunctionsBuilder.py" />
    <Exec Command="echo MEOS Functions built successfully" />
</Target>
```

Listing 5.3: Automating the function builder

5.3.2 Type Mapping

When generating the file with all the MEOS functions, we need to take some parameters in considerations. Indeed, in the MEOS header file, not everything is a function, which means we need to analyze the file to extract only the functions, which is what we do with regular expressions, as explained above. However, we need to change the signature of the methods with corresponding C# types. For instance, the type `char*` in C will be a string in C#. Thanks to the regular expression that we use, we can separate a function's signature into several groups and process it to replace C types by corresponding C# types. Every type that remains a pointer will be transformed to `IntPtr`, as described in Section 5.2. Finally, we can add the `DLLImport` annotation as well as all the lines needed to wrap these functions into a static class. The final file looks as what is shown on Listing 5.4. Remember that the complete code is available in Annex "MEOS.NET Repository".

```
using System.Runtime.InteropServices;

namespace MEOS.API.Internal
{
    internal static class MEOSFunctions
    {
        private const string DllPath = @"API\\Internal\\libmeos.so";

        [DllImport(DllPath, CallingConvention = CallingConvention.Cdecl)]
        public static extern void meos_initialize(string tz_str);

        [DllImport(DllPath, CallingConvention = CallingConvention.Cdecl)]
        public static extern void meos_finalize();

        // ...
    }
}
```

Listing 5.4: Structure of the class which exposes MEOS functions

5.3.3 Wrapping the Functions

After the generation of the C# file with all the MEOS functions, we want to show how to expose the features of MEOS outside the namespace in a safe manner. In this proof of concept, the `Temporal geometry point` has been chosen for a simple example. Since the temporal type is represented as a pointer, we need to be able to manipulate that pointer indirectly. To do so, we need to have the pointer as a private member of our new class. Then, we create public methods on which we have full control on what we can or can not do with that pointer. An example of this was given in Chapter 4. Again, the complete code is available under the namespace "MEOS.NET.Types" in Annex "MEOS.NET Repository". Keep in mind that this work is still a proof of concept, which means this process has been done for a very small amount of functions in the unique goal to demonstrate the work is valuable.

5.4 Unit Testing

Now that we exposed methods to the end-user, we want to ensure our code is correct and robust. Of course, we could create a sample application but this is clearly sub-optimal. A good code is unit tested, which is what we are aiming to do. When it comes to .NET and C# programming, there are **three** main unit testing frameworks : **xUnit**, **NUnit** and **MSTest**. According to P. Strzelecki and M. Skublewska-Paszkowska in their article "Comparison of selected tools to perform unit tests" [16], each of these frameworks are performing good in parallel test execution while MSTest outperforms xUnit and NUnit in traditional, sequential execution. Obviously, performance is not the only criteria of choice. However, we want to implement simple tests since we desire to check whether the MEOS functions are correctly called and yielding the correct result, nothing more. In that case, we do not need to worry about more complex features that some unit testing frameworks could provide. Furthermore, **MSTest** is the official Microsoft framework for .NET unit testing and, consequently, is the one that has stronger ties to Microsoft's official documentation and resources. Lastly, MSTest has been chosen as the framework for this study due to my familiarity with it. This familiarity is a key motivating factor behind the selection of this framework. Note that the tests for the proof of concept have been implemented in the Annex "MEOS.NET Repository". These tests are executed with the CLI and the "dotnet test" command.

5.5 Sample Application

Now that our library has some functions implemented and wrapped and that these functions are tested, we will review a simple, straightforward example of what is doable with MEOS.NET proof of concept. First of all, let us define what will the application do. We will start by creating a list of temporal geometry points from hardcoded strings as well as two other temporal geometry point

from a string. We will iterate through the list of temporal geometry points and compare to one of the single temporal values created after to demonstrate that the equality operators have correctly been wrapped. The last temporal geometry point that was created will be exported as JSON to a file. The complete example is shown on Listing 5.5 (and its result on Listing 5.6) and available in the Annex "MEOS.NET Repository". It is clear that this example may not be the most useful in a real-world application. However, this example leads to the conclusion that this project is feasible and realistic.

```
using MEOS.NET.General;
using MEOS.NET.Types.General;

var timezone = "UTC";
MEOSLifecycle.Initialize(timezone);

var temporals = new List<TemporalGeometryPoint>()
{
    TemporalGeometryPoint.From("[POINT(1 5)@2021-05-02, POINT(12 2)@2021-06-02]"),
    TemporalGeometryPoint.From("POINT(11 3)@2023-08-06 01:45:00+00:00"),
    TemporalGeometryPoint.From("[POINT(35 12)@2023-01-01, POINT(36 14)@2023-01-02]"),
};

var tempSeqSet = TemporalGeometryPoint.From("Interp=Step;{[POINT(1 1)@2000-01-01, POINT(2 2)@2000
    -01-02],[POINT(3 3)@2000-01-03, POINT(3 3)@2000-01-04]}");
var reference = TemporalGeometryPoint.From("[POINT(1 5)@2021-05-02, POINT(12 2)@2021-06-02]");

for (int i = 0; i < temporals.Count; i++)
{
    var text = (temporals[i] == reference) ? "equal" : "not equal";
    Console.WriteLine($"The {i + 1}th temporal element is {text} to the reference temporal");
}

var path = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.UserProfile), "
    json_output.json");
File.WriteAllText(path, tempSeqSet.ToJson());

MEOSLifecycle.Terminate();
```

Listing 5.5: A complete example of MEOS.NET usage

```
The 1th temporal element is equal to the reference temporal
The 2th temporal element is not equal to the reference temporal
The 3th temporal element is not equal to the reference temporal
```

Listing 5.6: Output of the example on Listing 5.5. The JSON output is available in Annex "MEOS.NET Repository

5.6 Future work

It is needless to say that this practical work is not completed yet. Of course, this is a proof of concept that aims to be improved over time. First of all, the main concern would be to implement more MEOS functions and other temporal types in order to harness the full potential of MEOS. So it goes without saying that unit tests have to be fully implemented for those new functions and types, but also provide new tests for edge cases. Moreover, the final library would be able to abstract the life cycle of MEOS to the end user (such as the calls to initialize and finalize the library). Besides, a better error handling should be implemented to feature a bullet-proof library. Finally, a layer, built on top of MEOS.NET, has to be created to ensure a seamless database connection with MobilityDB.

Conclusions

In conclusion, this work provides a new option to handle mobility data, and this, in C#. Indeed, a proof of concept was developed and shows that it is possible to wrap MEOS, the core component of MobilityDB, in .NET.

Starting from the core analysis of MobilityDB and MEOS, we explored their capabilities, but also why MobilityDB is a viable option thanks to the space and time optimizations. Furthermore, we explained how the internal type system of MobilityDB was working. After such an analysis, we could focus ourselves on the .NET ecosystem, its versatility and why C# is a language of choice for such a project. Finally, we provided a proof of concept showing that .NET is totally suitable to handle spatio-temporal and MobilityDB efficiently.

Obviously, a myriad of technologies exist and we could have chosen another technology stack for this project. However, we could demonstrate that MobilityDB embeds several optimizations to reduce the size of the resulting data set, which is a significantly good asset in a data-driven world. On top of that, temporal data is relatively new in the database research field, and MobilityDB is significantly efficient among other alternatives that exist, such as SECONDO. Furthermore, C# is a modern, high-level, open-source and widely used programming language. Its community and the support from Microsoft makes it a great choice to attract new researchers and contributors to the domain of mobility data. All of this makes MobilityDB and C# great technologies to provide to the applications the tools to implement mobility and spatio-temporal data seamlessly.

Yet the practical work presented in this work is not ready for production. However, this project is still a proof of concept. There are many things coming in the future for this project such as more MEOS functions being implemented but also direct and seamless database connection. In conclusion, even if there is still a little way to go, this work is a step forward in the field of providing efficient tools to manage mobility data to the .NET community.

Bibliography

[1] A. Broniewski, M. I. Tirmizi, E. Zimányi, and M. Sakr. Using MobilityDB and Grafana for Aviation Trajectory Analysis. *Engineering proceedings*, 28(17):17–, 2023.

[2] S. Datardina. Calling C library DLLs from C#, 2005.

[3] FernUniversität Hagen. BerlinMOD, 2023. `https://secondo-database.github.io/BerlinMOD/BerlinMOD.html`, Last accessed on 2023-08-03.

[4] J. Godfrid, P. Radnic, A. Vaisman, and E. Zimányi. Analyzing public transport in the city of Buenos Aires with MobilityDB. *Public transport*, 14(2):287–321, 2022.

[5] B. L. Gorman. *Practical Entity Framework Database Access for Enterprise Applications*. Apress, Berkeley, CA, 1st ed. 2020. edition, 2020.

[6] Microsoft. Type marshalling, 2022. `https://learn.microsoft.com/en-us/dotnet/standard/native-interop/type-marshalling`, Last accessed on 2023-08-06.

[7] Microsoft. Microsoft .NET and .NET Core, 2023. `https://learn.microsoft.com/en-us/lifecycle/products/microsoft-net-and-net-core`, Last accessed on 2023-07-14.

[8] Microsoft. Microsoft .NET Framework, 2023. `https://learn.microsoft.com/en-us/lifecycle/products/microsoft-net-framework`, Last accessed on 2023-07-14.

[9] Microsoft. .NET and .NET Core Support Policy, 2023. `https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core`, Last accessed on 2023-07-17.

[10] Microsoft. .NET Framework versions and dependencies, 2023. `https://learn.microsoft.com/en-us/dotnet/framework/migration-guide/versions-and-dependencies#net-framework-35`, Last accessed on 2023-07-14.

[11] Microsoft. What is .NET ?, 2023. `https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet`, Last accessed on 2023-07-11.

[12] Mobility Engine, Open Source. MEOS, 2023. `https://www.libmeos.com/`, Last accessed on 2023-08-03.

[13] C. Nagel. *Professional C# and .NET*. Wiley, 2021.

[14] PostGIS PSC & OSGeo . PostGIS, 2023. `http://postgis.net/`, Last accessed on 2023-07-30.

[15] StackOverflow. 2023 Developer Survey, 2023. `https://survey.stackoverflow.co/2023`, Last accessed on 2023-07-17.

[16] P. Strzelecki and M. Skublewska-Paszkowska. Comparison of selected tools to perform unit tests. *Journal of Computer Sciences Institute*, 9:334–339, 2018.

[17] N. Vermeir. *Introducing . NET 6: Getting Started with Blazor, MAUI, Windows App SDK, Desktop Development, and Containers*. Apress L. P, Berkeley, CA, 2022.

[18] R. Villela. *Pro . NET Framework with the Base Class Library: Understanding the Virtual Execution System and the Common Type System*. Apress L. P, Berkeley, CA, 2019.

[19] E. Zimányi, M. Sakr, and A. Lesuisse. MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS. *ACM Trans. Database Syst.*, 45(4), Dec. 2020.

[20] E. Zimanyi, M. Sakr, A. Lesuisse, and S. M. B. E. Mohamed. MobilityDB: A Mainstream Moving Object Database System. 2019-08-19.

Annexes

MEOS.NET Repository

This work is the support for a practical work called *MEOS.NET*, the wrapper for MEOS. The evolution of the project as well as the proof of concept can be seen on the Official Repository. The code of the proof of concept will forever be kept under a long-living branch named "release/poc". Further releases will be available on the main branch.

Proof of Concept's JSON Output

On Listing 7 is the (nicely formatted) output of the JSON of the example of Section 5.5.

```
{
  "type": "MovingGeomPoint",
  "crs": {
    "type": "Name",
    "properties": {
      "name": ""
    }
  },
  "bbox": [
    [1, 1],
    [3, 3]
  ],
  "period": {
    "begin": "2000-01-01T00:00:00+00",
    "end": "2000-01-04T00:00:00+00"
  },
  "sequences": [
    {
      "coordinates": [
        [1, 1],
        [2, 2]
      ],
      "datetimes": ["2000-01-01T00:00:00+00", "2000-01-02T00:00:00+00"],
      "lower_inc": true,
      "upper_inc": true
    },
    {
      "coordinates": [
        [3, 3],
        [3, 3]
      ],
      "datetimes": ["2000-01-03T00:00:00+00", "2000-01-04T00:00:00+00"],
      "lower_inc": true,
      "upper_inc": true
    }
  ],
  "interpolation": "Step"
}
```

Listing 7: The JSON output of the proof of concept's example