ULB FACULTÉ DES SCIENCES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Manipulation of GTFS periodic trajectories in MobilityDB

Mémoire présenté en vue de l'obtention du diplôme
de Scientifique en informatique à finalité spécialisée (MA-INFO)

Szymon SWIRYDOWICZ

Superviseur
Professeur Esteban Zimányi

Service
The Computer & Decision Engineering (CoDE) department

Année académique
2023 - 2024

Abstract

Effective management of transport data is essential for optimizing services, routes, and infrastructure in today's cities. MobilityDB, an open-source moving object extension of the popular PostgreSQL database management system, facilitates the management and analysis of such spatio-temporal trajectories. Previous research has used MobilityDB to analyse General Transit Feed Specification (GTFS) trajectories, which describe schedules of public transportation companies. An important property of these schedules is their weekly periodic repetition, which has not yet been utilized by the MobilityDB platform. Therefore, the aim of this study is to integrate periodic movements into MobilityDB. We start by exploring similar works in storing periodicities in databases and their extraction from data. We follow by adjusting the current data structure implementation to support periodic objects as well as implementing supporting methods for handling periodicity. Furthermore, we also analyse and compare the use of the periodic implementation with Brussels' public transportation (STIB/MIVB) GTFS data with the current non-periodic approach. Despite a few limitations and assumptions enforced by the regular nature of periodic movements, our results show interesting potential, notably in reducing the storage size of MobilityDB GTFS spatio-temporal sequences.

Keywords: moving object databases; periodic movement; MobilityDB; General Transit Feed Specification (GTFS)

Acknowledgement

First and foremost, I would like to thank my supervisor, Professor Esteban Zimányi, for his patience, trust, support, and remarkable availability throughout the course of this master's thesis.

I would also like to thank PhD student Maxime Schoemans for his advice regarding the periodic implementation and help with QGIS.

Lastly, I am thankful to my family and friends for their continuous support.

List of Abbreviations

CE(S)T	Central European (Summer) Time
DMBS	Database Management System
DST	Daylight Saving Time
GTFS	General Transit Feed Specification
MOD	Moving Object Database
PM	Pattern Mining
PPM	Periodic Pattern Mining
PSTDM	Periodic Spatio-temporal Data Mining
ST	Spatio-temporal
STIB-MIVB	Brussels Intercommunal Transport Company
SQL	Structured Query Language
TZ	Time Zone
ULB	Université libre de Bruxelles
UTC	Coordinated Universal Time

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This chapter introduces the topic, structure, and goals of this master's thesis.

1.1 Context

The subject of this master's thesis is to better handle periodic movements in MobilityDB, with a particular focus on General Transit Feed Specification (GTFS) transport schedule data. MobilityDB is an open-source geospatial trajectory data management and analysis platform, implemented as a PostgreSQL and PostGIS extension.

1.2 Motivation and Objectives

Thanks to the rapid development of GPS-supporting devices such as smartphones, vast amounts of trajectory data are being collected. These trajectories can represent different types of movements: a person's daily displacements, a public bus route, or even the migration of wildlife. Such movements often exhibit periodic patterns, which can be described as actions that are repeated at regular time intervals.

The goal will be to analyse how periodic information can be detected and extracted from spatio-temporal data, and how it can be utilized within a moving object database. Such information may help analyse the data semantically, drawing meaningful conclusions. Also, these patterns may help better handle the data and greatly reduce its storage size, which is increasingly important in the current Big Data era. Further details about MobilityDB and periodic objects will be given in further chapters.

Regarding pedagogical and scientific contributions of the chosen subject, it not only makes the student explore database-related scientific research, but also allows them to dive into and contributing to a real open-source project.

1.3 Contributions

- Summarize the state of the art in handling periodicities in spatio-temporal databases.
- Explain how periodic sequences can be integrated into the open-source MobilityDB ecosystem.
- Illustrate how periodicities can be utilized for the GTFS standard, with a focus on data from Brussels' STIB/MIVB public transport company.

1.4 Thesis Overview

This master's thesis is organized as follows.

- Chapter 2 introduces the reader to moving object databases, in particular to MobilityDB, as well as to the GTFS file standard.
- Chapter 3 describes the current research on periodic movements, exploring topics such as integration of periodic trajectories into databases or extracting of periodic patterns from data.
- Chapter 4 discusses how periodic movements can be implemented in MobilityDB, clearly listing different problematics that need to be considered for a correct implementation, as well as different operations that might be performed on such movements.
- Chapter 5 explores the effectiveness of the periodic implementation with the use case of STIB/MIVB GTFS.
- Chapter 6 further discusses the future development of this work and concludes the content of this master's thesis.

Chapter 2

Background

In this chapter we introduce the reader to the database infrastructure and tools we have used and built upon during this master's thesis.

2.1 Database Management Systems

Databases allow us to store, structure and easily manipulate large amounts of information. In order to manipulate that data, we use a Database Management System (DBMS). It is software that allows users to manipulate information through a relatively simple interface, while performing underlying optimizations to speed up data retrieval and improve storage efficiency. Most popular database management systems include names such as Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and MongoDB[1].

Relational data

There exist different paradigms and types of DBMS that change the approach of how the data is manipulated; in the scope of this paper, we will focus on relational database management systems (RDBMS). As its name suggests, the data model focuses on creating relations between data by organizing them into tables: each row contains one entity, or more formally, a tuple, and each related column describes a given entity by detailing its attributes. The whole table forms a relation[2]. For instance, Table 2.1 below groups together information about the latitude and longitude coordinates of STIB's stops[3] in a relational table format.

[1] https://db-engines.com/en/
[2] https://db-engines.com/en/article/Relational+DBMS?ref=RDBMS
[3] https://data.stib-mivb.brussels/explore/

ID (string)	name (string)	latitude (float)	longitude (float)
0631	GARE DU MIDI	50.836497	4.337946
3265	DELTA	50.818949	4.407743
3513	ULB	50.813926	4.384942
5462F	ULB	50.813657	4.384332

Table 2.1: Example "stib_stops" relational table.

SQL

As tables serve to organize and structure stored data, users use a database language to add, retrieve, remove, or, in general, manage specific information. The most well-established and standard query language used for relational data is SQL[4], which stands for Structured Query Language. One of its strengths, beside being powerful and expressive, is its readability and simplicity. For instance, executing the following SQL query

```
1 SELECT name, latitude, longitude
2 FROM stib_stops
3 WHERE ID = '3265'
```

on the "stib_stops" Table 2.1 will yield

```
1 DELTA, 50.818949, 4.407743
```

as it fetches tuples from the `stib_stops` table, applies a restriction based on the ID, and then selects the requested name, latitude, and longitude information.

As there is much more to explore on the subject, further explanations on relational databases and query languages are beyond the scope of this paper.

2.1.1 Spatial Databases

Spatial databases allow the manipulation of spatial data. Such data might be geometric, such as a simple point, line or any other form in a Cartesian space, or geographic, such as position obtained through a GPS (Global Positioning System) signal. One could argue that these data types could be abstracted in an ordinary relational database using just a couple of floats or strings. However, spatial database systems often not only define a spatial type system but also implement numerous optimizations, special indexes for fast data retrieval, and, most importantly for the end-user, a set of spatial operations and functions. These functions can, for instance, measure the distance between two points, draw a trajectory from a set of points, and check if trajectories cross a certain region.

[4] https://www.iso.org/standard/63555.html

2.1.2 Moving Object Databases

Moving objects databases (MOD) deal with any kind of object that moves. This could be a running person, bus, or bike travelling around Brussels, migration of a group of birds, or even displacement of hurricanes, clouds, lakes, or planets. In general, any continuous trajectory $[(l_1, t_1), (l_2, t_2), \ldots, (l_n, t_n)]$ combining both location points l_i and timestamps t_i.

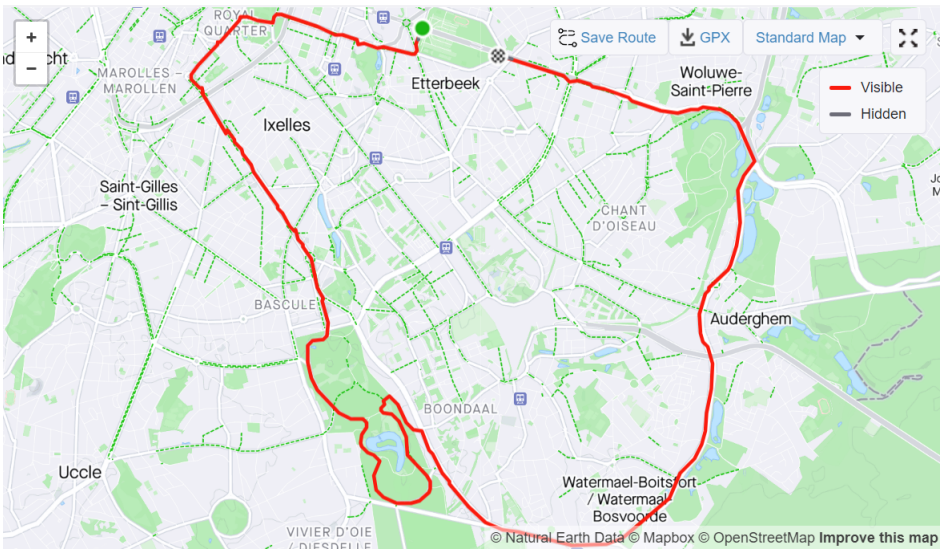

Figure 2.1: Example GPS trajectory.

There was an explosion of research in moving object databases around the 2000s, but little data to make effective use of that research [43]. With the relatively recent web expansion and lots of technologies to track such data, such as GPS trackers integrated into cars or our phones, what is missing are actual tools to analyse all that data.

Spatio-Temporal Data Types Different types of spatio-temporal data and representations exist, depending on the application and needs one is working with. Below are some examples, notably cited in [3, 37]:

- **Event data**: event points composed of both a spatial and time position. E.g., place and time when a crime occurred.
- **Trajectory data**: paths of objects moving through space over time. E.g., package route tracing.
- **Point reference data**: continuous spatio-temporal field. E.g., temperature differences over a field.
- **Raster data**: similar to point reference data; however, the locations and times of the points are fixed.

E.g., air quality measurement with fixed sensors, or a video, which is basically a sequence of images.

2.2 PostgreSQL

PostgreSQL[5] is an object-relational database system initially released in 1996 under a free and open-source licence, which means that the software and its code are publicly and freely available.

2.2.1 Extensible Platform

An important advantage of the platform's design is that it is extensible[6]. Database extensibility allows developing additional features without modifying the core code of the DBMS, and even adding them without restarting the running database service. It acts like an additional layer of functionality that has access to all the underlying layers.

Most relational database systems use what we call *system catalogues* to store information about databases, tables, or their columns: essentially a table of tables. Extensibility of PostgreSQL is based on the idea that its catalogues also store information such as data types, functions, operators, etc., allowing users to modify them as needed.

```
1 CREATE FUNCTION add_numbers(x integer, y integer)
2 RETURNS integer as $$
3     SELECT x + y;
4 $$ LANGUAGE SQL;
```

```
1 SELECT add_numbers(2, 2);
2 -- add_numbers| 4
```

Moreover, it also provides the ability to dynamically load user-written code. The following example loads a function named `Add_numbers`

```
1 CREATE FUNCTION add_numbers(integer, integer)
2     RETURNS integer
3     AS 'MODULE_PATHNAME', 'Add_numbers'
4     LANGUAGE C STRICT;
```

which is defined in an external file using the C-language; it receives two integer arguments as input and returns an integer back to PostgreSQL.

[5] https://www.postgresql.org/
[6] https://www.postgresql.org/docs/current/extend-how.html

```
PG_FUNCTION_INFO_V1(Add_numbers);

Datum
Add_numbers(PG_FUNCTION_ARGS)
{
    int32 x = PG_GETARG_INT32(0);
    int32 y = PG_GETARG_INT32(1);
    int32 result = x + y;
    PG_RETURN_INT32(result);
}
```

2.2.2 PostGIS

An important example of a spatial database system is PostGIS[7] (Geographic Information System). Initially released in 2001, PostGIS is an open-source PostgreSQL extension that adds support for geographic objects.

Geometry vs. Geography Two important data types PostGIS proposes are `geometry` and `geography`[8]. The former operates using Cartesian coordinates, which can be used to execute operations on a 2D plane or map. Among geometric types, we can find (i) *points*, (ii) *linestrings* which are paths obtained through point orderings, and (iii) *polygons* in order to represent areas. The latter, geography, uses spherical coordinates that take into account the properties of Earth's surface (not 2D). For example, it may be used to measure the distance between two coordinate locations with additional accuracy.

Some interesting PostGIS functions include:

- `ST_MakePoint`: transforms coordinates into internal geometry Point representation.
- `ST_SetSRID`: sets the SRID for a geometry. SRID stands for Spatial Reference System Identifier and is used to ensure spatial data is interpreted correctly depending on the specified geographic or projected coordinate system.
- `ST_DWithin`: returns a boolean indicating whether the geometries are within a given distance.
- `ST_Intersects`: returns a boolean indicating whether the geometries have any points in common.

[7] https://postgis.net/
[8] https://postgis.net/workshops/postgis-intro/geography.html

- `ST_AsText`: returns the Well-Known Text representation of a geometry, i.e., transforms the geometry into a human-readable format.

2.3 MobilityDB

MobilityDB [43] is an open-source geospatial trajectory data management and analysis platform developed by the Computer & Decision Engineering Department of the Université libre de Bruxelles (ULB)[9]. It notably includes a series of data types to organize spatio-temporal data, and functions to manage and analyse them.

It is important to note that it is implemented as a PostgreSQL and PostGIS extension. Extending PostgreSQL not only allows it to make use of the base relational system, but also greatly facilitates integration with the vast PostgreSQL ecosystem.

2.3.1 Comparison with PostGIS

One of the biggest advantages of MobilityDB in comparison to PostGIS is that it allows encapsulating a complete trajectory (both space and time) into a single object. Whereas PostGIS's type system relies on `geometry` and `geography`, MobilityDB implements `tgeompoint` and `tgeogpoint`, respectively, which group the former into spatio-temporal trajectories and allow an efficient manipulation of these objects.

Describing this in PostGIS would require using one tuple per trajectory point, along with additional columns to store the time information. Therefore, working on trajectories would require complex SQL queries that would first need to fetch all the individual point tuples. Moreover, as highlighted by [16], many intermediate points can be easily discarded thanks to the continuity properties of trajectories. They report that in their experiments on public transportation data, they managed to reduce the dataset from 500 MB/day to 5 MB/day thanks to MobilityDB, which is a massive 99% storage reduction.

2.3.2 MEOS

MobilityDB's core functionalities operate around the MEOS[10] C-language library that provides an API for manipulating spatio-temporal data. These two projects are highly connected as MEOS's development comes from the desire to keep the MobilityDB project separate from the PostgreSQL software, in order to allow other projects to be built upon it more easily (e.g., PyMEOS[11]) and to allow MEOS to work with other DBMSs such as MySQL in the future.

[9] https://mobilitydb.com/project.html
[10] https://www.libmeos.org/
[11] https://github.com/MobilityDB/PyMEOS

MEOS works with MobilityDB (PostgreSQL) as follows: MobilityDB wraps and transforms all the necessary PostgreSQL-related C code so that the equivalent MEOS API functions can be executed independently of the database codebase.

Building on our previous number example (cf. 2.2.1), we can illustrate this with the following code that calls the equivalent `meos_add_numbers()`[12] function, defined in plain C, within the MEOS library:

```
1   PG_FUNCTION_INFO_V1(Add_numbers);
2
3   Datum
4   Add_numbers(PG_FUNCTION_ARGS)
5   {
6       int32 x = PG_GETARG_INT32(0);
7       int32 y = PG_GETARG_INT32(1);
8       int32 result = meos_add_numbers(x, y);
9       PG_RETURN_INT32(result);
10  }
```

For simplicity, throughout most of this paper, we will often use MobilityDB to refer to both MobilityDB and MEOS interchangeably, except when otherwise specified.

2.3.3 Core Components

MobilityDB code revolves around temporal data, allowing the creation of moving objects. It combines (i) a base type, which represents the moving element, and (ii) a subtype, which changes how to value evolves over time.

Temporal Base Types

- `tbool` (temporal Boolean): e.g., evolution of a traffic light's state over time.
- `tint` (temporal integer): e.g., evolution of the world's human population.
- `tfloat` (temporal float): e.g., change in the speed of a bike.
- `ttext` (temporal text): e.g., text logs of messages displayed on an LED billboard.
- `tgeompoint` (temporal geometry point): e.g., evolution of an on-screen pixel.
- `tgeogpoint` (temporal geography point): e.g., location and time of a spacecraft travelling through outer space.

[12] Not an actual MEOS function. Given purely to illustrate the connection between MobilityDB and MEOS.

The first four base types are based on PostgreSQL's `bool`, `int`, `float`, and `text` types whereas the latter two are extensions of the basic `geometry` and `geography` types implemented by PostGIS and restricted to 2D or 3D points.

Temporal Subtypes

(i) Instant The smallest unit. Represents a value at a specific timestamp[13], separated by the *at* sign (@). The type of the value depends on the previously explained base type.

```
SELECT tfloat '25@2024-03-06 21:30:00';
```

(ii) Sequence Forms a trajectory by grouping instants into a single object. As these instants form a continuous movement, their timestamps must be provided in increasing order. Sequences can be split into three categories depending on how values are interpolated, as illustrated by Figure 2.2.

```
-- discrete
SELECT tfloat '{0@2024-03-01, 5@2024-03-06, 10@2024-04-11}';
```

```
-- linear
SELECT tfloat '(0@2024-03-01, 5@2024-03-06, 10@2024-03-11]';
```

```
-- step
SELECT tfloat 'Interp=Step; (0@2024-03-01, 5@2024-03-06, 10@2024-04-11]';
```

For instance, a linear interpolation allows obtaining intermediate values as if they evolved linearly.

```
SELECT valueAtTimestamp(
  tfloat '(0@2024-03-01, 5@2024-03-06, 10@2024-03-11]',
  '2024-03-03'
);
-- valueattimestamp|2
```

Note that `tbool`, `tint`, and `ttext` base types do not support linear interpolation due to how their types are defined.

Moreover, when the interpolation is either linear or stepwise, sequences are normalized. This means that any instant that does not change the direction of movement can be removed from the sequence, as illustrated by Figure 2.3.

[13]Note that timestamps in this paper are ordered from largest to smallest time unit, e.g., YYYY-MM-DD.

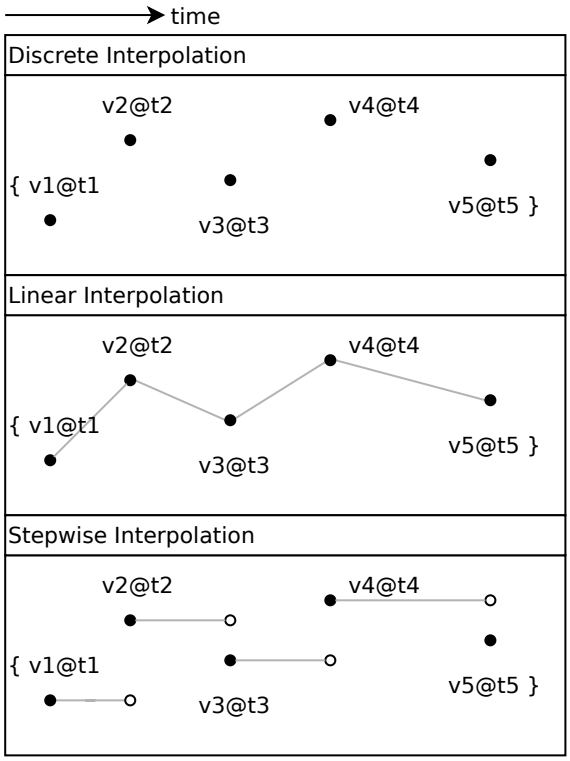

Figure 2.2: MEOS TSequence interpolation. [43]

```
1 SELECT tfloat '(0@2024-03-01, 5@2024-03-06, 10@2024-03-11]';
2 -- tfloat|(0@2024-03-01, 10@2024-03-11]
```

(iii) Sequence Set Groups sequences into a single object. As sequences are meant to be merged into a continuous movement by interpolating intermediate values, the primary purpose of sequence sets is to introduce time gaps where the value is undefined. Similarly to sequences, all instants within a sequence set must follow an increasing order, no matter to which sequence they belong.

```
1 SELECT ttext '{[Delta@2024-03-01 08:00:00, ULB@2024-03-01 08:30:00],
2 [Buyl@2024-03-01 08:45:00, VUB@2024-03-01 09:00:00]}';
```

Bounding Boxes

MobilityDB also defines bounding boxes which, as their name suggests, enclose both the value and time dimensions of each temporal object. A bounding box is defined for each temporal object and is mainly used for optimization purposes in operations and indexing, as it provides a summary of the extent of temporal movement. It is similar to PostGIS's `ST_Extent(geometry)`[14], which returns the bounding box of a geometry, but with an additional box for the time dimension.

[14] https://postgis.net/docs/ST_Extent.html

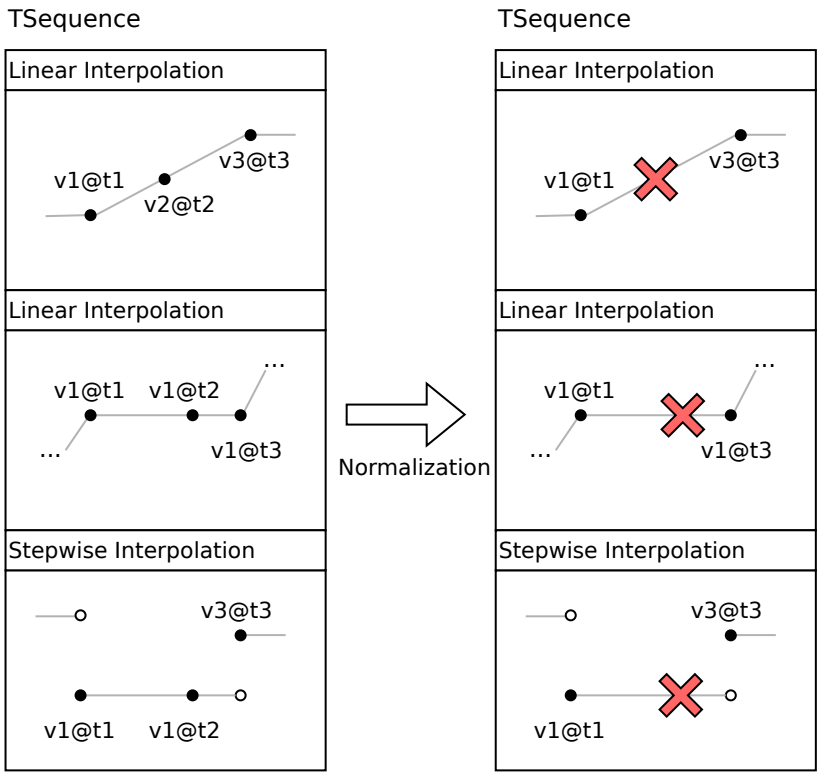

Figure 2.3: MEOS TSequence normalization. [43].

Bounding boxes come in two main types: `tbox` (temporal box) and `stbox` (spatio-temporal box). Intuitively, the former is defined using a numeric span, while the latter using 2D or 3D coordinate tuples, where the first and second tuples respectively define the minimal and maximal values.

```
1 SELECT tbox 'TBOXFLOAT XT([20.0, 30.0], [2001-01-01, 2001-01-02])';
2 SELECT stbox 'STBOX ZT(((1.0, 2.0, 3.0), (1.0, 2.0, 3.0)), [2001-01-01,
    2001-01-03])';
```

2.3.4 Operations

Besides providing an efficient way to store trajectories, MobilityDB also comes with a rich set of operations allowing to handle spatio-temporal data, including:

- **Accessors and Modifications**: functions such as `getValues`, `duration` and `startTimestamp` for accessing, and `insert` or `merge` for modifying temporal data.
- **Transformations**: operations such as `shiftTime` and `scaleTime`, or similarly `shiftValue` and `scaleValue`.
- **Restrictions**: function such as `atGeometry` or `atValues` allow data filtering.

- **Comparison**: temporal and spatial comparisons, such as *always or ever equal*, *smaller*, or *overlap*.

- **Spatial**: functions such as `nearestApproachInstant` to get the instant that is the closest to a given location, or `eDwithin` which applies ST_DWithin to a whole sequence.

- **Other**: additional operations such as simplification techniques to reduce the number of points in a trajectory and similarity measures such as *Fréchet distance* and *Dynamic Time Warping* (DTW) for comparing spatio-temporal data.

2.3.5 Visualization

Visualization is crucial for analysing geospatial data. QGIS[15] is a popular and open-source software that supports interactive viewing and editing of such data. Extended by the Move[16] plugin, it is possible to query MobilityDB databases and display the results with QGIS as further illustrated in 5.1.6. Moreover, QGIS's temporal navigation toolbar allows users to explore the trajectories over time, making it easier to understand object movement.

2.4 GTFS

The General Transit Feed Specification or GTFS[17] is an open standard format used to store and distribute information related to public transportation scheduling[18]. Adopted by many countries and public transit providers, it allows for easily sharing transit data, as well as develop various applications[19] including trip planning, predictions, monitoring, and much more. It is split into two categories: (i) GTFS Static or Schedule and (ii) GTFS Realtime.

GTFS Static GTFS Static consists of a collection of CSV (Comma-separated values) files, each storing information specific to an aspect of public transportation such as routes or schedules. Despite being a standard, it is rather loosely defined, in the sense that while some files are required, others might be optional, recommended or conditionally required/forbidden.

[15] https://www.qgis.org/en/site/
[16] https://github.com/mschoema/move
[17] https://gtfs.org/
[18] https://developers.google.com/transit/gtfs/examples/gtfs-feed
[19] https://gtfs.org/resources/apps/

Below we give an overview of the most pertinent GTFS files provided by STIB/MIVB[20], which is the *Brussels Intercommunal Transport Company*, as they will be the main case study of this paper (cf. Chapter 5).

- **Agency**: general information about the transport agency.

```
agency_id, agency_name, agency_url, agency_timezone, ...
STIB-MIVB, STIB, "http://www.stib-mivb.be", Europe/Brussels, ...
```

- **Routes**: information about transportation lines.

```
route_id, route_short_name, route_long_name, ...
63, 71, "DE BROUCKERE - DELTA", ...
```

- **Trips**: links where and when the vehicles should ride.

```
route_id, service_id, trip_id, trip_headsign, direction_id, shape_id,
    ...
63, 247134002, 115713844247134002, "DELTA", 0, 071b0196, ...
```

- **Stops**: information about each stop such as name or location (latitude, longitude).

```
stop_id, stop_name, stop_lat, stop_lon, ...
3514, "CIM. D'IXELLES", 50.815966, 4.390704, ...
```

- **Stop Times**: specifies the time of arrival at a stop.

```
trip_id, arrival_time, departure_time, stop_id, stop_sequence, ...
115713854247134002, 07:38:00, 07:38:00, 3514, 16, ...
```

- **Calendar**: specifies days of week when the service will be available (binary) as well as the date span of the service.

```
service_id, monday, tuesday, ... , sunday, start_date, end_date
247134002, 1, 1, ..., 0, 20221017, 20221109
```

- **Calendar Dates**: specifies exceptions in the regular service schedule (e.g., holiday); a 1 specifies a date that is added and a 2 a date that is removed.

```
service_id, date, exception_type
247134002, 20221020, 2
```

[20] https://www.stib-mivb.be/

- **Shapes**: provides the spatial geometry of transport routes, i.e., the spatial coordinate path which the vehicles follow.

```
shape_id, shape_pt_lat, shape_pt_lon, shape_pt_sequence
071b0196, 50.849673, 4.352683, 10001
```

Other optional files, not dealt with by STIB/MIVB GTFS but still interesting, include:

- **Frequencies**: allows trips to operate on a regular headway rather than a fixed daily schedule.

```
trip_id, start_time, end_time, headway_secs, exact_time
```

  Note that the use of frequency specification would be practical for our work, as periodic motion is directly specified by the format. However, as highlighted by [5]: (i) only a few transportation companies use this file format, and (ii) it is uncertain whether this is the optimal way of compressing the data.

- **Transfers**: specifies rules for making connections at transfer points between routes.

```
from_stop_id, to_stop_id,
from_route_id, to_route_id,
from_trip_id, to_trip_id,
transfer_type, min_transfer_time
```

  Although transfers can be deduced from stop times and stop proximity, this allows overriding these calculations by manually specifying possible connections.

GTFS Realtime GTFS Realtime provides continuous and live updates about the current situation of the transport network. It can include information such as (i) the position of vehicles, (ii) stop or route changes, (iii) unexpected events, (iv) delays, or (v) cancellations.

2.4.1 GTFS with MobilityDB

Authors of [16] successfully imported and analysed both Static and Realtime GTFS data of the Buenos Aires public transportation system using MobilityDB.

The import process varies between Static and Realtime, as the former consists of structured and pre-generated timetable text files, while the latter is retrieved dynamically as JSON responses from an API updated approximately every 30 seconds.

Also note that due to various possible formats of GTFS, their entire process cannot be simply generalized to any GTFS.

In order to import the static part, we can separate the process into three parts. First, pre-processing might be required in order to find arrival time anomalies, which cause problems for MobilityDB trajectory interpolation, and to remove unused information. Second, that data is copied into SQL tables depending on GTFS calendar specifications. Finally, a script generates actual MobilityDB data from the imported data by generating actual geometries and interpolating intermediate trip points, resulting in complete `tgeompoint` structures.

In comparison with the static part, the real-time part requires additional pre-processing due to the GPS-captured nature of the data. Indeed, imprecision and real-time errors might create outliers or incoherent points that need to be fixed or filtered. Moreover, due to imprecision and especially due to the infrequent sampling of trips, the obtained points require map-matching to ensure that generated trajectories follow actual routes rather than being "*as the crow flies*". Note that this can be done using methods directly provided by MobilityDB.

Moreover, they have demonstrated the usability of MobilityDB by performing various queries such as checking the average speed of vehicles, computing the shortest distances to routes, or analysing delays. Lastly, they pointed out an interesting direction for the future development of MobilityDB, which could focus on periodic movements and repetitions.

> *"GTFS Static represents periodic movement data that are valid during a certain time interval [7]. The current approach used in this paper requires to "instantiate" these periodic data to represent each actual occurrence. For instance, a service occurring every Monday at 8 am will be replicated for each Monday in the period of analysis. To avoid this, it is planned to implement in MobilityDB a new data type to account for periodic movements."* [16]

Following this direction, the goals of our work will involve the exploration of periodic data. We will investigate what can be achieved and how it can be implemented within MobilityDB, with a particular focus on the use case for public transport GTFS sequences.

Chapter 3

Periodicity and State of the Art

This chapter introduces the reader to periodic movements and summarizes the current state of the literature on the subject. We explore the research from two main perspectives: (i) storing periodicity in Moving Object Databases, (ii) extracting and mining periodic patterns from data.

3.1 Introduction to Periodic Patterns

Thanks to wide and rapid technological developments of tracking systems and telecommunications, we now have access to enormous quantities of spatio-temporal data, gathered from diverse domains [3, 8, 19, 25, 26, 37, 38] et al. Analysis of all that data might help in different applications including:

- Better transportation and traffic handling.
- Detection of unusual trajectories (e.g., in aviation or crime prevention).
- Climate prediction and forecasting.
- Understanding animal movement.
- Health care, neuroscience, and epidemiology.
- Social media.

A common property of spatio-temporal sequences that will particularly interest us in this paper is periodic behaviour. Put simply, we will be looking for repetitions in trajectories that occur at more or less the same locations over regular time intervals. A simple example is a public transport timetable, which is repeated for several days and weeks. In fact, summarizing long movements through repeating behaviours can provide valuable insights on studied data, revealing common patterns, helping to better understand the future regarding predictions, changes, or anomaly detection. In general, it also allows us to more efficiently compress and store data [25, 28].

To better understand what periodicity exactly is, we will start by analysing how it is defined in some papers we will later explore. The goal is mainly to demonstrate the variety of definitions and to get some ideas about how periodicity can be represented.

First, we will take a look at the actual definitions. According to the Encyclopædia Britannica[1], *periodic motion* is defined as a movement repeated in equal time intervals. That time interval, defining the length of the cycle, is designated as a *period*. Additionally, we might also define the *frequency*, which is reciprocal to the period and represents the number of repetitions within a time unit. For instance, the Earth orbits the Solar system within a one-year period, and reciprocally, with a one orbit frequency per year.

As discussed further by [33] (cf. 3.2.4), the term *periodicity* can have several meanings. It could be an event that:

- Repeats itself exactly after an interval, e.g., each Monday at 08:00.
- Repeats itself within an interval, e.g., once a week but not necessarily on the same day.
- Will eventually repeat in the future, e.g., the event will repeat itself, but we do not know when.

They describe these definitions as *strong*, *near*, and *weak* periodicities, respectively. Note that we will mainly focus on implementing strong periodicities in this paper.

From a mathematical perspective, [20] (cf. 3.2.3) defines the periodicity based on sets: a set $S \subseteq \mathbb{Z}$ is said *periodic* if there exist a positive integer p (or period) such that for all $n \in \mathbb{Z}$ and for all $i \in S$, $i + np \in S$. In other words, values repeat, as after p steps we will always obtain the same value.

In [18] (cf. 3.3.1), periodicity is defined on the basis of patterns:

- *Patterns* are non-empty sequences $s = s_1 \dots s_p$ over $(2^L - \{\emptyset\}) \cup \{*\}$[2], where L is a set of features (i.e., properties, locations, ...) and $*$ the character that matches any feature (wildcard).
- *Period* is defined as the length of the pattern s. For instance, we can find the pattern "a*b" in "acbaceaebd" with a period 3.

[1] https://www.britannica.com/science/periodic-motion
[2] Reminder: 2^S is a notation for the set of all subsets of S.

- *Frequent partial periodic pattern* is essentially defined as a pattern that repeats itself modulo each period and is present inside a sequence a number of times greater than a certain constant threshold.

We can find a similar definition in [28] (cf. 3.3.2), but with the focus on spatial trajectories:

- *Sequence* `S` of spatial locations $\{l_0, l_1, \ldots, l_{n-1}\}$ (movement).
- *Period* is an integer $T << n$ representing, for instance, a week or a month.
- *Periodic segment* $\{l_i, l_i{+}1, \ldots, l_{i+T-1}\}$ as a sub-sequence of `S` such that i mod T equals 0 (i.e., the segment repeats itself after T locations).
- *Periodic pattern* `P` as a sequence $r_0 r_1 \ldots r_{T-1}$ with r_i being either a spatial region (cluster) or $*$ (wildcard, any). In other words, they search for periodicities within spatial regions enclosing several locations rather than within the locations themselves.
- *Frequent pattern* is a pattern that appears at least *min_support* times in a sequence `S`.

Furthermore, in [26] (cf. 3.3.3) the definition not only includes time information at locations, but also defines periodicity using probabilities:

- An interpolated location sequence $LOC = loc_1 loc_2 \cdots loc_n$ with position $loc_i = (x_i, y_i, time_i)$.
- A set of *reference spots* $O = o_1, o_2, \ldots, o_d$ which are the most visited places in the trajectory.
- A *periodic behaviour* defined as a pair $< T, P >$ with
  - T the regular time interval (period).
  - P a probability distribution matrix of probabilities that the moving object is at reference spot o_i at a given timestamp $t_k \leq T$.

Finally, [31, 38] classify periodic patterns into the following categories:

- (i) Full: every position in the pattern is periodic.
- (i) Partial: mixes periodic patterns with non-periodic ones (i.e., wildcards are permitted).
- (ii) Perfect: pattern must occur every period.

- (ii) Imperfect: pattern might occur every period.
- (iii) Synchronous: no misalignment nor noise in-between patterns.
- (iii) Asynchronous: misalignment or noise in-between patterns is permitted.

3.2 Literature Review: Periodicity in DBMS

In this section, we will analyse existing research on handling periodic sequences in databases. We will focus on how to efficiently represent, store, and query sequences that exhibit repetitive patterns over time.

3.2.1 Nested Periodic Tree

The authors of [6, 7], to our knowledge, have made the most recent attempt at implementing periodic sequences into moving object databases.

The idea of their approach is to construct a tree structure (see Figure 3.1) which we will call the periodic tree. It not only allows storing both repeating and basic movements together, but also allows repetitions to be nested.

The periodic tree is read depth-first (from root to leaves, exploring left child first) and can have four different nodes:

Node T Tree root. Anchors time to an arbitrary timestamp (`2000-01-01 08:00:00` in the example) to which the whole sequence will be relative to.

Node L Link node. A pointer to a basic movement unit or the unit itself. In loose terms, a movement unit can be described as going from point A to point B. The time in these movements is relative, which means that it is described by a duration rather than an actual timestamp. Note that such units can be empty, to describe the lack of movement during a certain duration.

Node R Repetition node. Stores how many times its children should repeat.

Node C Composite node. Stores L and R nodes. The order of its children describes the temporal order of the sequence as the movements are relative to the T node.

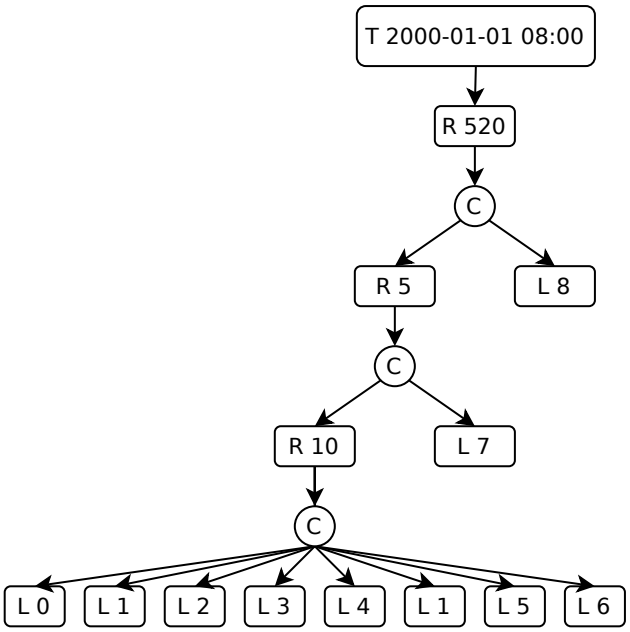

Figure 3.1: An example of the complete tree representation. [7]

For example, Figure 3.1 presents public transportation vehicle data, which, starting from 2000-01-01 08:00, for 520 weeks (∼10 years), 5 days a week, 10 times a day, will perform the L_0 to L_6 movements, where L_6 is a small break at a terminus stop. Higher up in the tree, L_7 and L_8 correspond to staying at the terminus during the night and during the weekend, respectively.

The authors have implemented these ideas using `SECONDO` [17], which is a popular research extensible-DBMS for testing and prototyping database concepts with spatio-temporal support. The data structure is composed of a few fixed-size attributes like anchor or total duration as well as separately stored arrays for each type of tree node. Note that as it is a DMBS, arrays are used instead of tree pointers in order to avoid frequent data serialization.

Additionally, an algorithm for detecting repetition and constructing the periodic tree in an integer sequence is presented. However, as stated by the authors themselves, it is not able to handle either time or position noise. As noise is omnipresent in practice due to notably GPS measurement imprecision, the algorithm would need to be adapted for noise, or the input sequences would need to be pre-processed.

The whole implementation was evaluated using two data sets: one containing periodic and generated train data (best-case scenario) and the other real-world GPS data without any a priori periodicity (worst-case scenario). They report that the total size of the first (periodic) data set, when converted to the periodic tree representation, dropped remarkably from 429.56 MB to 0.37 MB (−99.9%) in 117 seconds. The size of the second data set (non-periodic) increased from 35.58 MB to 39.43 MB (+10.8%) in 223 seconds.

Two operations were also compared: `atinstant()` (restricts to a time instant)

and `trajectory()` (returns 2D line of movement) based on their time cost, as illustrated in Table 3.1, showing clear improvements for the periodic train example and similar performance for the non-periodic approach.

Data set	**Train**	**GPS**
atinstant (flat)	6.7	0.58
atinstant (periodic)	0.0023	0.33
trajectory (flat)	149.00	11.4
trajectory (periodic)	0.25	11.2

Table 3.1: Periodic tree operation time cost (in seconds). [7]

3.2.2 Parametric Rectangles

Authors of [9] introduce *parametric rectangles*. These rectangles have the form

$$< X_1, \ldots, X_d, T >$$

where T is a real interval representing the time instances, and X_i is a function from $\mathbb{R}$ to these real intervals, representing the spatial extent of the rectangle. An example could be a boat moving $5m/s$ east (towards positive x) during $t = 0$ to $t = 10$ with an initial position of $(9 \leq x \leq 19, 10 \leq y \leq 20)$, which would be described by the parametric rectangle $r =< [5t + 9, 5t + 19], [10, 20], [0, 10] >$, i.e., $< x, y, t >$.

In MobilityDB, these could be compared to temporal bounding boxes, but where the value dimension is no longer static and instead changes over time according to some mathematical function.

That definition is further extended to *periodic parametric rectangles* by simply adding a non-negative integer constant named the *period*. It can be intuitively described by a function of the form $f(t \bmod p)$ with t representing the time, p the period and *mod* the modulus operator, or in other words, where the time cycles.

Later in [29], they extend the structure to also support acyclic movements which combine periodic with linear movement. In simpler words, a cyclic movement always repeats the same position and velocity every period, whereas acyclic movement allows the position to change each period as long as the general form of the pattern is preserved.

They also describe an extended relational algebra query language, describing relations such as projection, selection, union, or intersection, which they prove can be evaluated in polynomial time.

To conclude, they provide an interesting manner of approximating the movement of periodic objects using bounding boxes whose dimensions are functions of time. The

use of rectangles also has the advantage of enclosing spatial coordinate imprecisions, which are often encountered in periodic trajectories, which can suffer from slight value changes from one period to another.

However, as highlighted by [7] (cf. 3.2.1), it should be noted that the *Periodic Parametric Rectangle* approach does not support nesting of periodic movements. Reciprocally, the *Periodic Tree* approach discussed previously does not support acyclic movements.

3.2.3 Multislices

Authors of [20] explore efficient and concise encoding of public transport schedules. Although transport schedules follow periodic behaviours, there are irregularities that appear frequently, such as cancelled or additional trips due to special occasions like strikes or holidays, making a concise periodic representation more difficult.

They divide trips into two data structures: a relative trip and a repeating trip. The former stores ordered tuples of (position, time) where the time is stored as the offset from the departure time at the starting station. The latter stores temporal repetition information regarding only the starting station, such that the repetitions at other stations can be inferred when joined with a relative trip.

Temporal repetition is compressed into structures named *slices*, which define a time granularity hierarchy, intuitively resembling an interval representation. These are stored as a tuple list $\lambda = G_1X_1, \ldots, G_dX_d$ with G_l being a time granularity and X_l a selector defined as an integer set. For instance, the slice $\lambda_1 = (yea\{24\}, wee\{0-25\}, day\{0-4\}, hou\{7\}, min\{0, 25, 55\})$ represents intuitively a trajectory ranging in the first 26 weeks of 2024, from Monday to Friday, at 7am with 0, 25 and 55 starting minutes. Note that although a month (mth) time granularity is possible, in general the hierarchy follows $(yea, wee, day, hou, min)$, as months are irregular, as we will discuss later in 4.1.1.

Slices can be combined into *multislices*; a tree structure allowing more complex trajectories, with the largest time units lying higher towards the root. For instance, we could add a node for $day\{5-6\}$, i.e., the weekend, which would have a child node $hou\{8-9\}$ instead of $hou\{7\}$, starting the trips at a later hour. Also note that the *tree* can be easily stored as relations in a relational DBMS, as each slice can be stored as a separate table row, with individual time granularities being organized by ID into separate tables.

Finally, they define *mixed recurrence* as a pair of (λ_r, λ_e) with λ_r representing rules, which follow previous trip definitions, and λ_e exceptions, which represent gaps or cancelled trips. This introduces semi-periodic behaviour as the result set is defined as the difference $\lambda_r \setminus \lambda_e$. Similarly, a tree can be constructed using the union

operation on different (λ_r, λ_e) pairs.

Moreover, in [21] the authors build upon these ideas by developing compression methods. Since they prove that the optimal compression is NP-hard, they provide a heuristic method with promising and near-optimal results, running in $O(d^2 n \log n)$ time, where d is the hierarchy depth and n is the recurrence size.

However, despite merge operations, they do not discuss actual (periodic) operations utilizing these multislices, focusing mainly on the compression of trips into a compact structure.

A similar quote was made my the authors of [5] where they state:

> *"In that work, the main contribution is to encode single trips as efficiently as possible (regarding operation days, holidays etc.) rather than extracting periodicities in the first place."*

Moreover, in their paper, they assume that real-world schedules do not change temporally from one trip to another and only assume the delay to be present at the start time of sequences. As we will discuss later in this paper with STIB/MIVB schedules (cf. 5.1.8), travel time between stops might change from one trip to another. These delays are notably caused by road traffic and are often incorporated directly into the schedule in order to provide more accurate arrival times. Although not studied, it could possibly be achieved by a series of similar relative trips and multislices. However, describing consecutive trips using separate tree structures might have a negative impact on performance.

Regarding MobilityDB, these ideas could be naively implemented by representing the relative trip as a temporal sequence, and storing the mixed recurrence using a separate tree structure.

3.2.4 Periodic SQL

Several authors also discuss the handling of periodicities in temporal databases (not necessarily spatio-temporal), putting their focus on periodic relational algebra rather than the efficient encoding. Although we will not discuss these papers in detail, they still present some interesting ideas, especially regarding user queries.

In [33], the authors formalize periodicity as well as periodic query expressiveness for temporal databases, building on the concepts of LRPs (linear repeating point) i.e., periodic sets of points of the form $an + b$, and of temporal calculus, i.e., an extension of traditional relational calculus with temporal aspects. They do so by considering (i) strongly periodic sets of time, and (ii) nearly periodic sets of time. The former assumes points are *eventually equally distanced from each other* (e.g.,

exactly each Monday), whereas the latter leverages the equality to allow points to be within a certain limit range from each other (e.g., once every week).

They also discuss how to add periodicity to query languages such as notably TSQL2, which in brief extends SQL with more efficient temporal data handling. An example of periodic query they provide is

```
SELECT empl_name
FROM attend-2 A
WHERE meeting = 'CRC'
  AND periodic(week, Monday, T)
  AND BEGIN(A) <= T < END(A);
```

which intuitively queries the name of employees that attended any Monday meeting.

In [32], the authors focus on defining a high-level language for *symbolic* user-defined periodicity, in comparison with more mathematical expressions. They define a periodic event as a pair $<< d_1, d_2 >, p >$. The ds represent the timestamps between which the event occurs, which not only serve as a time filter but also as an anchor point for synchronizing the periodicity. The p represents the periodicity, being a user-defined symbolic expression such as "*on each Saturday*". Additionally, they compare symbolic and mathematical approaches, where they criticize the use of LRPs, especially regarding weeks and months being asynchronous. As months are irregular (cf. 4.1.1), they highlight that the minimal repetition period between them is 28 years (336 months or 1461 weeks). Thus, it would require 336 LRP formulae to describe the relation "the first Monday of each month" as they require to explicitly list all terms of the union.

Additionally, although PostgreSQL does not natively support TSQL2, we might simply achieve some query ideas discussed above using PostgreSQL's time functions. For instance, a query such as "*happened on the 1st Monday of a month*" could be written as:

```
SELECT
    DATE_TRUNC('month', CURRENT_DATE) + i_day * '1 day'::interval AS date
FROM
  generate_series(0, 6) AS i_day
WHERE
  EXTRACT(dow FROM
    DATE_TRUNC('month', CURRENT_DATE) + i_day * '1 day'::interval
  ) = 1;
```

Which

- generates a series of 7 numbers, one for each day of the week, with `generate_series`,
- truncates the argument date to 1st day of month with `DATE_TRUNC('month', CURRENT_DATE)`,
- adds the day offset to the truncated date with `i_day * '1 day'::interval`,
- checks if the date is a Monday using `EXTRACT(dow FROM date) = 1` (0=Sunday, 1=Monday, ...),
- and finally selects the date that is the first Monday of the month.

3.3 Literature Review: Periodic Pattern Mining

As defined by IBM[3], data mining is the process of uncovering patterns and other valuable information. In recent years, especially with the development of big data, a significant amount of research has developed around spatio-temporal data mining, which, as its name suggests, is a subset of data mining that focuses on addressing spatio-temporal problems. Among these we can find clustering, predictive learning, frequent pattern mining, anomaly detection, change detection, and relationship mining [3]. The process that will interest us in this section is periodic pattern mining, or PPM in short, which focuses on discovering patterns that focus on time intervals and their regularity.

Note that there are two other spatio-temporal pattern mining processes often encountered in the literature that are related to periodic pattern mining, which are frequent pattern mining and sequential pattern mining. The former focuses on how frequently a pattern occurs and its co-occurrence, i.e., in terms of the quantity of patterns, and the latter on the order of events. Both of them might be useful for periodic pattern mining. For instance, frequent pattern mining may help to find the most frequent movement patterns and sequential pattern mining to identify movement progression. Together they may be a foundation for further analysis of the periodicity or regularity of these patterns. Also note that in a more general terminology, frequent pattern category will often encompass the other two.

Uncertainties

Before diving into the actual methods related to mining, we will examine several aspects of periodic patterns that should be considered.

[3] https://www.ibm.com/cloud/learn/data-mining

While looking for periodicity in position data, GPS for instance, the measured subject will in most cases rarely be measured at exactly the two same locations and at exactly the same time [28]. Even though the subject might follow the same routes regularly, the measured position and timestamps of points and trajectories might slightly vary from one another. Thus, despite presenting obvious similarities and sharing clear semantic periodicity, if periodicity mining procedures only consider evenly sampled, regular, perfectly matching data, very few patterns will be discovered. Additionally, real-world spatio-temporal data might contain imprecision and gaps due to malfunction of measurement tools, or simply due to irregular measurements. This might result in incorrect periodicity detection.

As previously explained in 2.3, MobilityDB interpolates intermediate points in order to ensure the continuity of trajectories. However, in terms of data analysis, it is still important to note the existence of uncertainties; noisy, uneven and incomplete raw data might draw wrong conclusions. For example, if we only have two points while measuring a person's trajectory: one in the morning and the second in the evening, in both of which the subject is located at their home, linear interpolation might not be able to draw correct conclusions regarding this person's displacements during the rest of the day [25].

Following, as highlighted by [3], traditional pattern mining techniques are not suitable for finding patterns in these trajectories due to the imprecise nature of spatial information. As we will explore later in this section, it will require us to consider slightly different approaches, such as summarizing or enclosing movements into regions.

3.3.1 PPM in Time Series

Before exploring the discovery of periodicity in spatio-temporal data, we will briefly examine the mining process in the simpler case of time series, as most spatio-temporal approaches aim to reduce the problem to this case.

A popular approach was developed in [18] where the authors propose an approach called *max-subpattern hit-set* for partial periodic patterns. The method relies on an influential property in data mining and discovered in [1], known as the Apriori property. It ensures that if a sub-pattern is not frequent, then the pattern itself is not frequent either. This allows for quickly discarding uninteresting patterns rather than searching through all possible patterns. This property can be extended to periodic patterns by simply restricting it to a given period.

Following this, they propose a simple 1-period algorithm consisting of two main steps. (i) Start with the smallest possible patterns (1-patterns) of period p. Accumulate the frequency count within each whole period and eliminate patterns that

are below a threshold frequency. (ii) Repeat for patterns of increasing size i up to p, and terminate when the i-size pattern set is empty.

However, as highlighted by the authors, although in association rule mining, where the Apriori property is commonly used, the number of item sets decreases rapidly, that number decreases slowly in the case of partial periodicity mining. Therefore, they extend it to the *max-subpattern hit-set* algorithm which combines the first step of the previous algorithm with *max-patterns*, which are defined as the maximal patterns which can be generated from a set of frequent 1-patterns. In short, the idea is to scan and count subpatterns of the max-subpattern in different period segments. The hit set is the set of all found frequent subpatterns, and frequent patterns can be further derived from that set (Apriori). This reduces the number of scans from p to 2. They further develop these ideas to support multiple periods.

3.3.2 `STPMine`

Spatio-temporal trajectories of moving objects are vulnerable to potential irregularities and noise, complicating periodicity detection. The idea presented in [11, 28] is to divide the space in which the moving object travels into a region grid. This is illustrated in Figure 3.2, where three trajectories are presented, each occurring on a different day but with the same start and end points which are labelled by the `A` and `G` regions. This results in sequences AACCCG, AACBDG, and AAACHG and the partial periodic patterns AA***G, AAC**G, AA*C*G with supports 3, 2, 2 respectively.

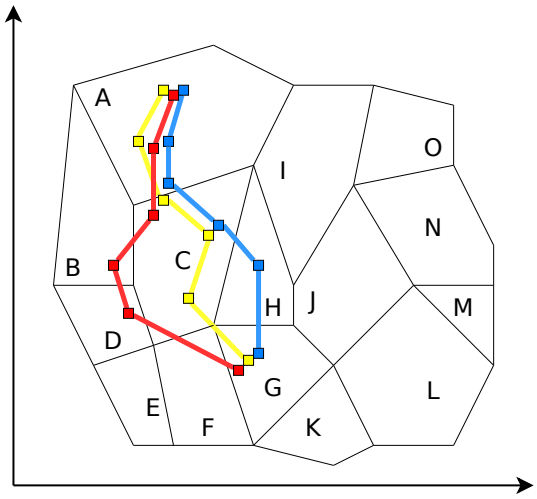

Figure 3.2: Periodic patterns with pre-defined spatial regions. [11, 28]

The goal is to simplify the trajectories and allow a discrete representation of patterns, in such a way that the trajectories can be described only using sequences of regions' labels.

Such regions could be, for example, districts, network cells or just any random grid. However, using such regions can lead to undesirable patterns; trajectory points close to each other may fall into completely different regions if these are defined too small, and inversely for distant points if the regions are not dense enough. Consequently, the region definition can be done automatically, which the authors have implemented through data clustering.

Data clustering consist in grouping similar objects together in groups named clusters. In comparison with the region partition from Figure 3.2, Figure 3.3 illustrates this with five round-shaped clusters.

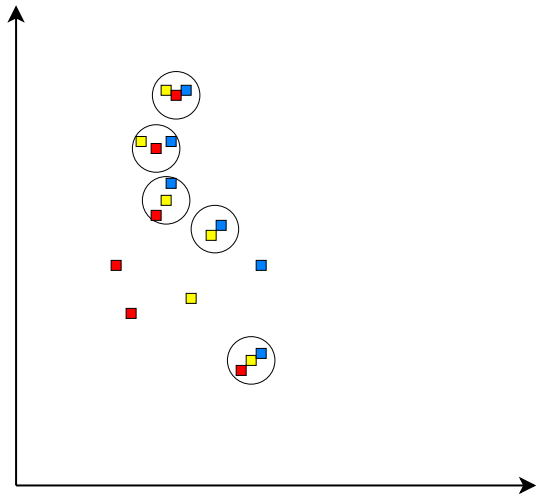

Figure 3.3: Illustration of trajectory clustering. [28]

There are various clustering algorithms in the literature. Below, we list three that are often encountered in PPM:

- DBSCAN [15]: Density-Based Spatial Clustering of Applications with Noise. As its name suggests, it clusters points based on their density. It is notably used by `STPMine`.

- HDBSCAN [10]: Extends DBSCAN with hierarchical clustering, allowing the identification of clusters with varying densities. Used by `HCSPPM` (cf. 3.3.4).

- TRACLUS [24]: *Partition-and-group* framework for clustering trajectories. Divides trajectories into characteristic points, which are then further clustered.

Finally, following this idea, the clusters can be considered as sequences, which reduces the problem to finding periodic patterns in time series data using algorithms as the *Max-Subpattern Hit Set* proposed in [18] (cf. 3.3.1).

3.3.3 `Periodica`

The `Periodica` algorithm is presented in [26, 27]. Its goal is to mine periodic behaviours using a location-based method, and it is composed of two main stages: the period detection and periodic behaviour mining.

Period Detection

The *period* in *periodic behaviours* is in simple words the interval of time it takes for a behaviour to repeat. This could for instance be a day, weak, year or any other time interval. In certain applications, it may be sufficient for the period to be specified by the user. However, in other cases that period might not be known. Moreover, a movement can hide periodic behaviours following different hidden periods (e.g., one every hour and another every day), which even further motivates automatic period detection.

The first part of finding periods, is to find interesting *reference spots*. *Reference spots* consist of important, frequently visited zones that will allow to reduce the whole movement into a collection of specific places. This could for example be a home, workplace and/or university if we track a student's behaviour, or a set of large zones such as a city and countryside if we track bird displacement. Summarizing the movement to such spots allows changing the spatial point sequence (trajectory) into a binary sequence $b_1 b_2 \dots b_t \dots b_n$: is the subject inside ($b_t = 1$) or outside ($b_t = 0$) the reference spot at a specific timestamp t. Simply checking when the moving object is present in a particular spatial zone can greatly facilitate period detection as uninteresting spatial noise is discarded.

In brief, their approach of finding the reference spots consists of discretizing the space into cells and computing their point density, since the reference spots are considered to have more points than other places.

Finally, in order to detect periods, which is done for each reference spot separately, common methods for finding periodicity in time series are used. For instance, a mixture of Fourier transformation (identification of frequencies) and autocorrelation (detection of repeating patterns) can be used as further discussed by [36]. The process has an estimated time complexity of $O(n \log n)$ per spot, where n corresponds to the length of each binary sequence.

Behaviour Mining

In their paper, the periodic behaviour H of a segment set I is modelled by both a specific period T and a probability distribution matrix P which encodes the movement with its uncertainties. This is illustrated in Table 3.2 where the period is 24 hours and where two reference spots (dorm and office) were identified.

	8:00	**9:00**	**10:00**	...	**17:00**	**18:00**	**19:00**
Dorm	0.9	0.2	0.1	...	0.2	0.7	0.8
Office	0.05	0.7	0.95	...	0.75	0.2	0.1
Unknown	0.05	0.1	0.05	...	0.05	0.1	0.1

Table 3.2: Example of daily periodic behaviour using a probability matrix. [26]

Another problem we should also note is that multiple periodic behaviours might be following the same period. For instance, a student might follow a strict study schedule every day but have a completely different daily behaviour during the holidays. In short, the proposed idea is to cluster these different days together based on their temporal location using a metric to compare the similarity of their probability distributions.

Once the reference spots and their periods are determined, spots with the same periods are combined. Next, a *symbolized movement sequence* $s_1, s_2, \ldots, s_i, \ldots, s_n$ is built, which allows telling to which reference spot (o_j) a movement's location belongs ($s_i = j$). The sequence is then divided into smaller sequences, where $I_k^j = i$ is used to denote the j^{th} segment and k^{th} timestamp and i^{th} reference spot. For instance, $I_9^5 = 2$ with a 24h period means that the object is at spot number 2, the 5^{th} day at 9h. At this point each I forms a single cluster. These are combined using an iterative cluster-combining algorithm which merges the closest clusters together by measuring the distances separating them. As periodic behaviours are described using probability matrices, the distance is computed between these probability distributions using for instance *Kullback-Leibler divergence* [22] which has the form $\sum P(x) log(\frac{P(x)}{Q(x)})$ (with P and Q being the compared distributions). The idea is that the smaller the distance between two probability distributions is, the more likely it is that the segments were generated from the same periodic behaviour. Their estimated time complexity of the algorithm is $O(m^2 \cdot T \cdot d + m^2 \cdot \log m)$, where T, d and m are respectively the period, the number of reference spots, and the number of segments.

Use Case

A similar structure to *Periodica* can be found in *"Spatiotemporal Periodical Pattern Mining in Traffic Data"* [19] where the authors analyse a speed dataset from highway monitoring stations. The algorithm has two stages: (i) periodic behaviour detection, and (ii) station clustering. Stage 1 discretizes speeds into a fixed number of levels, constructs boolean series for each level, and finds the period using Fourier transformation followed by autocorrelation. Then, it picks the prime period from across different levels and constructs categorical distribution matrices to find the

periodic behaviour. Finally, stage 2 clusters the stations based on matrix similarity by also using *Kullback-Leibler divergence*.

3.3.4 HCSPPM

Hierarchical clustering-based semantic period pattern mining, or in short HCSPPM, is a periodic pattern mining method proposed in [40]. Note that this paper extends previous works by the same authors where more details about semantics [41] and hierarchy [39] can be found.

Previous Approaches

The authors of this paper have listed a series of drawbacks of the previously explored approaches.

First, the fixed period approach (cf. 3.3.2). It requires the user to specify a period during which the algorithm should look for periodic patterns. While this might be sufficient for some applications, in general hidden periodic patterns might be missed outside the specified period. Also, the employed region division and clustering approaches ignore the temporal aspect of the trajectories.

Secondly, the reference spot approach (cf. 3.3.3). The clustering method that allows finding the reference spots focuses on points and does not take into account the temporality of objects in trajectory clustering.

Moreover, both require the input trajectories to be regularly sampled, and neither takes advantage of the hierarchical and semantic information.

Important Characteristics

Semantics An interesting approach is to take semantic information into account. In other words, rather than simply considering spatial data as meaningless geographies, the idea is to exploit the meaning behind the data. Thus, the trajectories considered in the paper are not only composed of position and time information, but also carry semantic information. This also applies to the area that a reference spot covers, which could for instance be a university.

The semantic information can, for instance, be retrieved from the OpenStreetMap[4] open licence map project. This allows obtaining different details, including the place's ID, name, type, and geometry.

Hierarchy Instead of working with single-level reference spots, the hierarchical nature of space can be considered. Hierarchical reference spots are composed of

[4] https://www.openstreetmap.org/

multiple layers. For instance, instead of considering a group of reference spots representing cities, we could merge them together into a state or city, or on the other hand, divide them into a collection of buildings each city comprises.

(Semantic) Stop Episodes Stop episodes are simply defined as places where people stay for a significant amount of time [42]. Combined with the semantic information, semantic stop episodes might reveal meaningful activities.

The general outline of their algorithm can be summarized by eight steps as shows Figure 3.4 below:

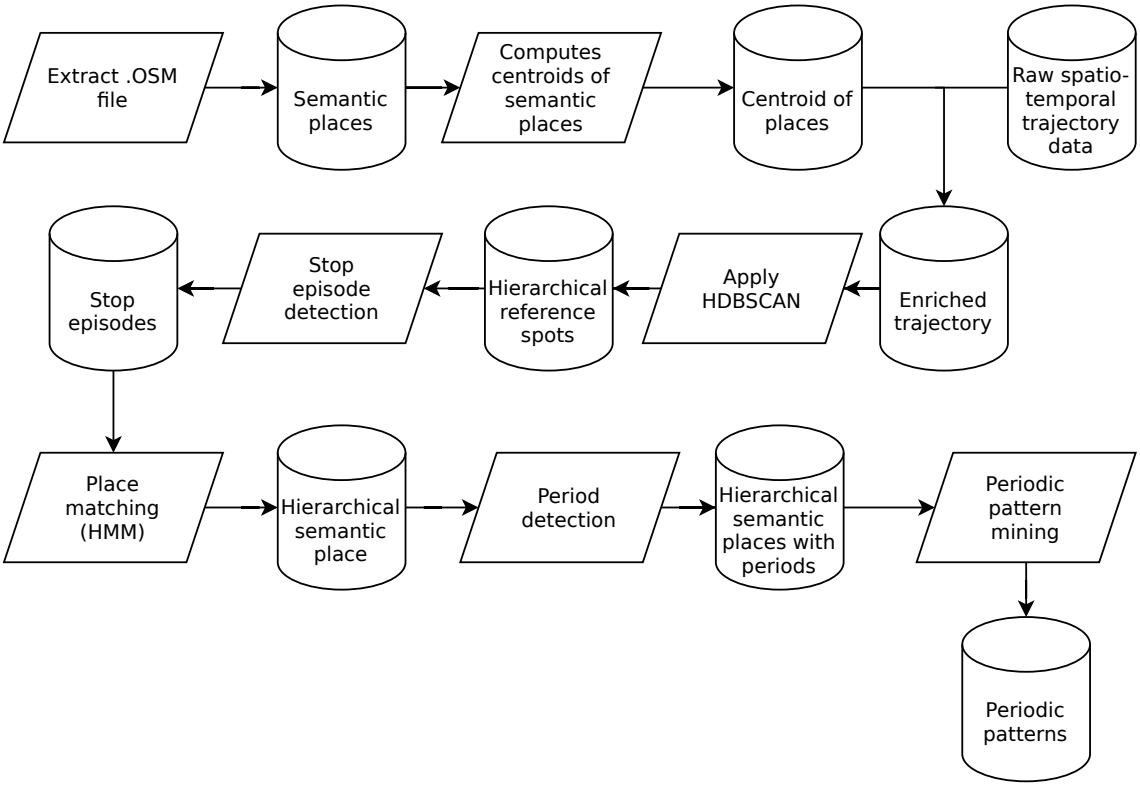

Figure 3.4: Framework of hierarchical semantic periodic pattern mining from spatio-temporal trajectories. [40]

Citation Review

The authors of a FGO-based approach (2022) [35] commented on the algorithm shown in Figure 3.4 in their paper, stating that it is "*computationally effective and identifies interesting hierarchical periodic patterns*" but they also cited some disadvantages of the paper, stating that "*hierarchical periodic patterns need to be visualized, and the experiments are not conducted using diverse datasets*".

In addition, in their own paper, they propose an ANN-based (artificial neural network), FGO (football game-optimized) TLBO (teaching-learning-based optimization) algorithm for periodic pattern mining. In short, it starts by transforming

the data into a matrix, then TLBO is applied for clustering purposes, and finally the ANN-FGO extracts the periodic patterns. It is interesting to note that they compared the results obtained by their ANN-FGO approach with those of HCSPPM using rail transit data. In their test environment, they obtained a time consumption and accuracy of around 25,000 seconds and 96% for their method, compared to 60,000 seconds and 88% for HCSPPM.

Chapter 4

Methodology and Implementation

This chapter explains the choices and designs made during this master's thesis to implement periodic sequences in the MobilityDB system. Moreover, we present some algorithms and structures that were implemented, as well as their importance and purpose.

Combining what we learned in Chapter 2 and 3, we will attempt to integrate periodic movements into MobilityDB. As MobilityDB is a project of an important size, one objective to keep in mind is to ensure our implementation does not break or overly complicate the development and usage of the current MobilityDB system. On the other hand, as our goal is to integrate the concept of periodicity specifically into MobilityDB, we want our implementation to work with and use as much of the existing framework as possible.

4.1 Relative Sequences

Our goal is to build a data structure that will be able to both (i) store a simplified and compressed version of a sequence with repetitions and (ii) decompress itself by repeating itself in time.

To achieve this, we will take inspiration from the concept of relative intervals, notably defined in [7] as:

$$RelInterval = (l, lc, rc) \mid l \in \underline{duration}, lc, rc \in \underline{bool}$$

Here, duration is intuitively defined as the integer distance between two timestamps (in milliseconds) and (lc, rc) describe end point inclusion.

This allows us to store the base sequence and reuse it for different timestamp values. It can be converted to an absolute interval by simply assigning a timestamp

t to the relative interval which is denoted in [7] as:

$$(RelInterval \leftarrow t) = ((l, lc, rc) \leftarrow t) = (t, t + l, lc, rc)$$

Note that we will call this process *anchoring*.

In order to port this idea into MobilityDB, we can develop it starting with the following `ttext` sequence as an example:

```
SELECT ttext '[A@2024-06-01, B@2024-06-02, C@2024-06-03, A@2024-06-04,
   B@2024-06-05, C@2024-06-06]';
```

It repeats the pattern "ABC" with a period of 3 days, with the repeating part being included between `2024-06-01` and `2024-06-03` timestamps. Instead of using `text@timestamp`, we could make the repeating part of the sequence relative using a `text@duration` format. The time part of each instant composing a sequence would be replaced by its duration relative to the first instant of that sequence. As displaying the duration in milliseconds would clearly be illegible for users, PostgreSQL Interval data type can be used instead:

```
SELECT ttext '[A@0, B@1 day, C@2 days]';
```

Note that the first interval value is 0 as it is relative to itself.

This concept can be easily integrated to the current Temporal implementation. The time part of a `TInstant` structure is defined by a `TimestampTz` which stands for a timestamp with time zone information. However, internally, a `TimestampTz` is defined using an `int64` i.e., a 64-bit integer. Similarly to a Unix Epoch Timestamp which is stored as the number of milliseconds (ms) since `1970-01-01 00:00:00 UTC`, timestamps in PostgreSQL, and by extension, MobilityDB, are stored as the number of microseconds (μs) since `2000-01-01 00:00:00 UTC`. Therefore, as both intervals and timestamps can be expressed using microseconds, the duration can be given directly to a `TInstant` as a timestamp without needing to redefine the data structure. This gives us this equivalent sequence:

```
SELECT ttext '[A@2000-01-01, B@2000-01-02, C@2000-01-03]';
```

Additionally, we can change the value-instant symbol '`@`' to '`#`' in order to clearly distinguish an ordinary temporal sequence from a relative sequence:

```
SELECT ttext '[A#2000-01-01, B#2000-01-02, C#2000-01-03]';
```

We can find similar ideas notably in [21] (cf. multislices 3.2.3), where they assume that each time granularity such as *yea*{7} starts with the `2000-01-01 00:00` instant.

4.1.1 Textual Representation

Year 2000 happens to be a round number and is especially easy to read due to its three trailing zeros, making it a good starting reference point for our relative sequence. And yet, we could extend the output formats of relative sequences to make it more readable to users depending on the period of repetition. We propose the following textual representations as a basis, which we will then discuss in terms of their advantages and disadvantages:

- Timestamp style (default)

```
'[A#2000-01-01 00:00:00, B#2000-01-01 10:00:00, ...]'
```

- Interval style

```
'[A#0 days 00:00:00, B#0 days 10:00:00, ...]'
```

- Per day

```
'[A#00:00:00, B#10:00:00, ...]'
```

- Per week

```
'[A#Monday 00:00:00, B#Monday 10:00:00, ...]'
```

- Per month

```
'[A#01 00:00:00, B#16 10:00:00, ...]'
```

- Per year

```
'[A#Jan 01 00:00:00, B#Mar 06 10:00:00, ...]'
```

Interval Style Details

Regarding the interval style, although the 'day' scale should be sufficient to describe a duration, Interval formatting also allows additional designators including months or years:

```
'C#6 years 5 mons 4 days 3 hours 2 mins 1 secs'
```

Alternatively, using the more compact ISO 8601 style with all designators:

```
'C#P7Y5M4DT3H2M1S'
```

Additionally, note that all consecutive interval-style instants are relative to the 2000 reference timestamp and not to each other.

Week Style: First Day of the Week

The Week style does not correspond to and does not restrict the user to the actual day of the week. In reality, January 1, 2000 happens to be a Saturday, but we simply follow the intuitive order and assume that the sequence starts with the first day of the week. As the sequence is supposed to be relative, it is up to the user to choose the appropriate day of the week as the start point for anchoring whenever a weekly periodicity is desired.

An important remark is the first day of the week might change depending on the local culture[1]. For instance, in most of Europe and North-Asia the first day of the week corresponds to a Monday, whereas it is a Sunday for North-America and South-Asia, and a Saturday for North-Africa. Thus, for the implementation, we might either assume that the first day of the week is:

- Sunday: following PostgreSQL[2].
- Monday[3]: follows (i) ULB/Belgium and mainly (ii) the ISO 8601[4] standard.
- Local: respects the local preferences depending on PostgreSQL locale[5].

Month Problems

Per month and per year textual representations seem to be a good intuitive choice for handling periodic sequences that repeat each month and year respectively. Nevertheless, it is crucial to recognize that months are not periodic in the mathematical sense. In fact, the number of days in a month in the Gregorian calendar varies between 28, 29, 30, and 31 days. February has either 28 or 29 days due to leap years, whereas other months have either 30 or 31 days. Similarly, years are not periodic due to the leap system as they can be either 365 or 366 days long. Therefore, although monthly reasoning seems intuitive for users, we cannot establish a single-period repeating pattern for months or years. For instance, a relative sequence that has a different value for each day of January (31 days), when anchored to the 1st of February, will in consequence overflow over March. Thus, we might consider the following approaches in order to handle this problem:

- Count using day intervals only and thus allow sequences to overflow.

```
1 SELECT timestamptz '2000-01-31' + interval '31 days';
2 -- 2000-03-02 00:00:00 (overflow)
```

[1] https://www.unicode.org/cldr/charts/42/supplemental/territory_information.html
[2] https://www.postgresql.org/docs/8.1/functions-datetime.html
[3] We will assume Monday as the first day of the week for the rest of this thesis.
[4] https://www.iso.org/iso-8601-date-and-time-format.html
[5] https://www.postgresql.org/docs/current/locale.html

Disadvantages:

  - Not intuitive for users as we intuitively expect from month-relative intervals to be enclosed within the month's span.

- Allow months to be trimmed. Values of removed days would either be lost or merged (e.g., mix, average, ...) into remaining days.

```
SELECT timestamptz '2000-01-31' + interval '1 month';
-- 2000-02-29 00:00:00 (trim)
```

Disadvantages:

  - Information might be lost.
  - Confusing behaviour if the sequence is not anchored to the 1st day of a month (per month) or to the 1st day of a year (per year).
  - Troublesome for operations (functions) that rely on periodic properties of sequences due to irregular period of months.

- Remove the possibility to create per month and per year sequences.

Likewise, interval representation suffers from similar problems whenever month or year designators are used:

```
'A#0, B#31 days, C#62 days'
-- A#00:00:00, B#31 days, C#62 days
'A#0, B#1 month, C#2 months'
-- A#00:00:00, B#31 days, C#60 days
```

Here, it is sufficient to restrict the highest possible interval unit to either days or weeks (as they are synchronous with days).

Similarly, MobilityDB internally defines the number of days per year as 365.25, and per month as 30 following ISO 8601 suggestions. Otherwise, more accurate values can be considered such as $365.2425/12 = 30.436875$. Although they are imprecise, these approximations are often sufficient for date/time related computations.

Furthermore, another imprecision factor might come from the leap second system. Additional seconds might be added or removed in order to reflect Earth's rotation, usually at the end of June and December, which might create days with 86,401 seconds instead of 86,400. However, as these changes are irregular and unpredictable, we will not take them into account for our periodic sequences.

Deferred Sequence Start

Previously, we defined that relative sequences are meant to represent the duration relative to the first element of the sequence, i.e., having the first element always starting at time 0. We will take as an example the following timetable sequence:

```
1 HERMAN-DEBROUX  # 00:00:00,
2 DELTA           # 00:04:00,
3 ARTS-LOI        # 00:14:00,
4 GARE DE L'OUEST # 00:25:00,
5 ERASME          # 00:37:00
```

The sequence can then be anchored to a timestamp such as '2024-06-01 08:30' and repeat itself with a '1 day' period.

Although this allows potentially reusing the sequence for different time values within the same day (e.g., start at 08:35, 8:40, 8:46, ...), in practice, public transportation schedules may change between instants from one sequence to another depending on the time of the day. Factors including traffic density or the number of passengers might increase delays in trajectories which are taken into account when creating timetables. For example, it might take 14 minutes to reach 'ARTS-LOI' in the morning during high density periods, and just 12 minutes during a calm period at night.

Instead of creating multiple relative sequences and storing the start time of each alongside them, it might be easier and more intuitive to create "semi-relative" sequences. These would allow the first instant of the sequence to be different from the 2000-01-01 00:00:00 reference point and include the time delay directly within the sequence.

```
1 HERMAN-DEBROUX  # 08:30:00,
2 DELTA           # 08:34:00,
3 ARTS-LOI        # 08:44:00,
4 GARE DE L'OUEST # 08:55:00,
5 ERASME          # 09:07:00
```

For anchoring, it would then be sufficient to input '2024-06-01' without the time part to obtain the equivalent result. Although it is equivalent to defining no movement during the `'[00:00:00, 08:30:00)'` interval as in:

```
1 HERMAN-DEBROUX  # 00:00:00,
2 HERMAN-DEBROUX  # 08:30:00,
3 DELTA           # 08:34:00,
4 ARTS-LOI        # 08:44:00,
5 GARE DE L'OUEST # 08:55:00,
6 ERASME          # 09:07:00
```

the main difference would concern the repetition point of the sequence. The repetition period of the relative sequence in `[00:00:00, 09:07:00]` is 9 hours 7 minutes whereas it remains only 37 minutes for the semi-relative in `[08:30:00, 09:07:00]`: thus the sequence might be repeated, for example, every hour. In summary, the anchoring point would remain the 2000-01-01 reference, while the period and repetition would be relative to the first instant of the sequence.

An important note regarding the GTFS format (cf. 2.4) is that it follows an hour system greater than 24 hours in order to account for overnight trips. For instance, a trip scheduled for '2024-06-01' might contain time values such as '27:59:59'. However, although the times might exceed 24 hours overnight, due to the periodic nature of the GTFS standard, the total duration of trips remains less than a day; otherwise, the trips would overlap with themselves between days. This reinforces the need for semi-relative sequences as (i) trips are supposed to repeat at least every day (period <24h) and (ii) most first trips begin approximatively after 05:00 in the morning and stop before 03:00.

Representation Overflow

Since we assume that a sequence does not need to start exactly at 2000-01-01 00:00:00, textual representation might cycle. This is a common scenario for overnight transport:

```
01 # Monday 08:00:00, -- 2000-01-01
...,
98 # Sunday 23:00:00, -- 2000-01-07
99 # Monday 01:00:00 -- 2000-01-08
```

Note that the second Monday does not correspond to the same date as the first Monday. As a reminder, Temporal requires instants to be ordered and ascending in time due to interpolation and continuity constraints. In order to distinguish between these two values, a simple solution is to suffix the sequence with a number n indicating the number of cycles (as in $i + np$).

```
01 # Monday 08:00:00,
...,
98 # Sunday 23:00:00,
99 # Monday 01:00:00+1W
```

Where for each overflow-prone style we have:

```
+[number]D --days
+[number]W --weeks
-- +[number]M --months
-- +[number]Y --years
```

However, note that the implementation of month and year overflows might be either inaccurate or troublesome, since, as discussed previously (cf. 4.1.1), they are irregular.

Time Zones

As discussed previously, timestamps in PostgreSQL and MobilityDB are relative to `2000-01-01 00:00:00 UTC`. UTC stands for Coordinated Universal Time and is a globally used reference time standard. In fact, other time zones are defined by their offset from UTC, such as CET (Central European Time) which corresponds to UTC+01:00 or 1 hour ahead of UTC.

An advantage of UTC is that it remains constant throughout the year and does not follow summer or winter times, also known as daylight saving time (DST). For instance, Central Europe distinguishes between CET (Central European Time) and CEST (Central European Summer Time), which switch between UTC+01:00 and UTC+02:00 on the last Sunday of March and the last Sunday of October respectively.

This can lead to misleading behaviour when dealing with relative sequences. To illustrate, we can suppose a user from Belgium inputs the following as their work start time.

```
Work # Monday 08:00:00
```

Which timestamp corresponds to:

```
SELECT '2000-01-01 08:00:00'::timestamptz;
-- timestamptz|2000-01-01 08:00:00+01
```

or directly in UTC:

```
SELECT '2000-01-01 08:00:00'::timestamptz AT TIME ZONE 'UTC';
-- 2000-01-01 07:00:00
```

However, when anchored or repeated during the summer, the sequence will result in:

```
Work # Monday 09:00:00+02
```

because of the CEST time zone.

```
SELECT '2000-06-01 07:00:00 UTC'::timestamptz AT TIME ZONE 'CEST';
```

As this is technically a correct conversion, in practice the user would usually prefer that the work starts at 08:00:00+02 rather than 09:00:00+02 as it follows their usual schedule or work routine.

Moreover, an important problem might appear whenever we mix time zones for anchoring. Suppose the instant:

```
Monday 10:00:00+01 --CET
```

which converted to internal UTC gives:

```
Monday 09:00:00+00 --UTC
```

When we want to anchor to:

```
2024-01-29 --CET
```

it is equivalent to:

```
2024-01-28 23:00:00+00 --UTC
```

Since we assume relative sequence to be relative to 2000-01-01 00:00:00 UTC, the total relative instant duration is equivalent to 9 hours, which gives:

```
2024-01-28 23:00:00+00 + 9 hours
= 2024-01-29 08:00:00+00 --UTC
```

which back to CET gives:

```
2024-01-30 09:00:00+01 --CET
```

From a CET-only perspective, intuitively we would expect the final instant to be one hour later i.e., at 10:00:00+01. Thus, we can either assume our users will always work using UTC i.e., as if they worked with `timestamp` (time zone independent) rather than `timestamptz`, or modify the reference point. In fact, rather than computing the duration regarding UTC, it could be computed vis-à-vis the used time zone, in this example to 2000-01-01 00:00:00+01 (CET). Thereby, the total duration in our previous example would be 10 hours regardless of the anchor's time zone. In the case where multiple time zones are mixed within a single sequence or for sequences in general, we could take the reference point to follow the time zone of the first instant, in order to avoid edge cases such as DST changes.

Nevertheless, `TimestampTz` does not allow us to extract the time zones from the first instant. In fact, `TimestampTz` input is directly converted to UTC, and then simply displayed using locale time zone information.

```
SHOW timezone;
-- TimeZone|Europe/Brussels
SELECT '2000-01-01 10:00:00+15'::timestamptz;
-- timestamptz|1999-12-31 20:00:00+01
SELECT ('2000-01-01 10:00:00+15'::timestamptz)::timestamp;
-- timestamp|1999-12-31 20:00:00
SELECT '2000-01-01 10:00:00+15'::timestamp;
-- timestamp|2000-01-01 10:00:00
```

Thus, for choosing the reference point we can follow one of these ideas:

- Choose 2000 at UTC. Safe option as it follows internal time as well as remains free of DST changes. However, as discussed previously, gives confusing behaviour when used with sequences not defined using UTC. To avoid confusion, only the Interval style might be kept as it directly describes a duration unlike date styles such as DAY or WEEK. Otherwise, the time zone might need to be ignored for instant input and output.
- Choose 2000 at locale time zone. Interesting option as it follows default `timestamptz` behaviour as well as user settings. However, similarly to choosing 2000 UTC, it struggles when sequences are defined using time zones different from the locale, or during DST changes. Moreover, the relative sequence duration might change whenever the user's location changes. In fact, this might produce inconsistent anchor results when sharing sequences between databases following different locales.
- Define reference point as first element of the sequence. However, goes against the deferred sequence ideas discussed previously (cf. 4.1.1).
- Input time zone as function argument. However, this is not user-friendly as it requires an understanding of internal implementation.

As the focus of this master's thesis is on the case study of STIB/MIVB GTFS, we choose to keep DAY and WEEK styles, which nicely represent GTFS sequences. In order to avoid possible confusion, we choose to remove all time zone information in relative sequences.

4.2 Periodicity

Now that we have narrowed the requirements for a relative sequence, we will describe how we can add additional information to transform it into a periodic sequence, i.e., that repeats regularly in time.

Parameters Although many approaches exist, we will first focus on adding the following ingredients:

- **Period** ≔ `Interval`. It is the time interval after which the sequence repeats, or in other words, the length of a cycle. As discussed previously (cf. 4.1.1), we define the period to be relative to the first element of the sequence rather than 2000-01-01 UTC. Moreover, we require that it must be greater than or equal to the duration of the relative sequence. Otherwise, the sequence would simply overlap with itself as it repeats. By default, we set the period to the duration of the given sequence.

- **Anchor span** := `TstzSpan`. Anchors the relative sequence in time, transforming it into an ordinary temporal sequence. Although traditionally this process requires only a single timestamp, we explicitly require it to be a timestamp span instead. As the anchoring process can be achieved using a simple MobilityDB `shiftTime` function, we rather focus this parameter to define the complete span for all the repetitions. In fact, and for obvious reasons, we cannot make periodic sequences repeat indefinitely in practice; an upper bound must be defined. Note that the upper bound could be defined using a tuple (t, r) with t being the anchor timestamp and r a positive integer defining the total number of repetitions. However, it is in general easier to reason about long-term periodic behaviours within a span range rather than with the number of repetitions. Moreover, using a span also makes the comparison of multiple periodic sequences easier. Whenever two periodic sequences have different periods, it is more convenient to compare both of them within the same time span, rather than manually fine-tuning their number of repetitions in order to make them correspond in length. Finally, the existing MobilityDB (timestamptz) span[6] data structure can be used. You may also note that it supports lower and upper bound inclusion.

Moreover, we might also consider additional parameters:

- **Number of repetitions** := `Integer`. Similarly to the periodic tree (cf. 3.2.1), it defines the maximum number of times the sequence should repeat itself. This parameter is optional if the end bound of the anchor span is defined, but it might help to better restrict the repetitions if the number is known beforehand.

- **Strict pattern** := `Boolean`. Defines whether the whole sequence must fit within the anchor span. If set to true, the last occurrence of the sequence will not appear in the complete sequence if the last time value is greater than the end bound of the `anchor span`, i.e., if the whole pattern does not fit within the anchor span. Otherwise, the last occurrence of the sequence will be simply trimmed such that all time values are within the `anchor span`. For example, it can be useful for vehicle trips. It might be wiser to not depart the trip at all if it cannot reach the destination within a defined time, rather than making it stop in the middle of nowhere.

[6] https://mobilitydb.github.io/MobilityDB/master/ch02.html

4.3 Data Structure

In order to integrate periodicity into MobilityDB sequences, we need to understand how temporal sequences are defined. `Temporal` is an abstract data type that serves as a common basis for all temporal types as illustrated by Figure 4.1. Its main attributes are `temptype` which defines a value's data type such as `tfloat`, `ttext`, or `tpoint`, as well as `subtype` which defines whether it is an instant, sequence, or sequence set.

Each `subtype` builds on top of another. First, `TInstant` is the smallest unit, storing a `TimestampTz t` as well as a `Datum`[7] value, which type depends on the `temptype`. Following, `TSequence` is built as an array of instants, storing individual `TInstant` structures. And similarly to `TSequence`, `TSequenceSet` stores `TSequence` sequences.

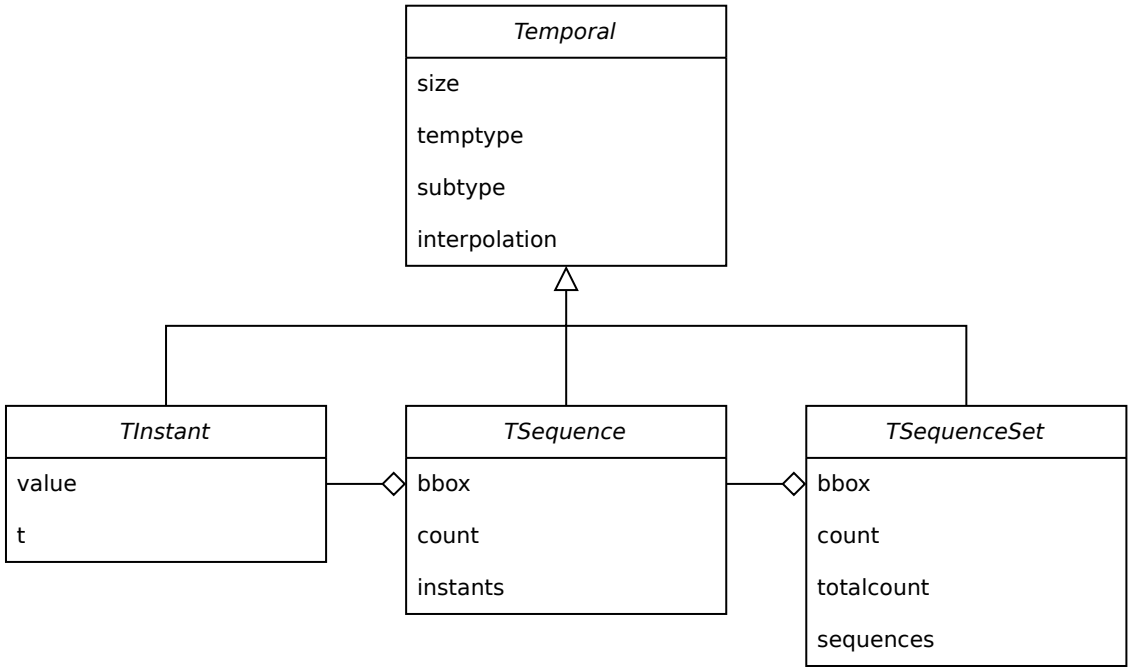

Figure 4.1: MEOS conceptual type hierarchy. [43]

4.3.1 Implementation Attempts

Following previous discussions, by simply assuming that `TInstant`'s time is relative to 2000-01-01 00:00:00, and that `Temporal` subtypes are built upon each other, we are able to create relative trajectories. In order to transform relative trajectories into periodic trajectories, i.e., trajectories that are meant to be repeated in time, we need to find a way to store previously defined periodic attributes alongside our sequences.

[7] Datum is a generic and internal PostgreSQL data type representation, similar to (void *) in the C-language.

Before explaining the chosen implementation approach, we will start by discussing some common implementation ideas and the reasons that make their development complicated.

Inheritance

Although MobilityDB codebase is written using the C procedural programming language, the Temporal data type follows an object-oriented programming (OOP) structure. Thus, the first idea for implementing periodicity was to define it based on the concept of inheritance. Periodic data types would inherit their equivalent Temporal data types, extending their functionalities with additional periodic attributes.

The main advantage of using inheritance is that it would allow simply casting between Periodic and Temporal structures. In consequence, a Periodic structure could be interpreted as an ordinary Temporal, therefore enabling the reuse of existing MobilityDB functions on periodic sequences for simple operations.

```
1 Periodic *per = ...;
2 Temporal *temp = (Temporal *) per;
```

However, this approach does not work in practice because Temporal is defined as a variable-length data structure as illustrated by Figure 4.2. Indeed, the size of `instants` in `TSequence` can change during program runtime, as users can freely extend sequences with new instants. As a result, whenever a Periodic type would be used in a function disguised as a Temporal, the periodic information could be overwritten by new instant data.

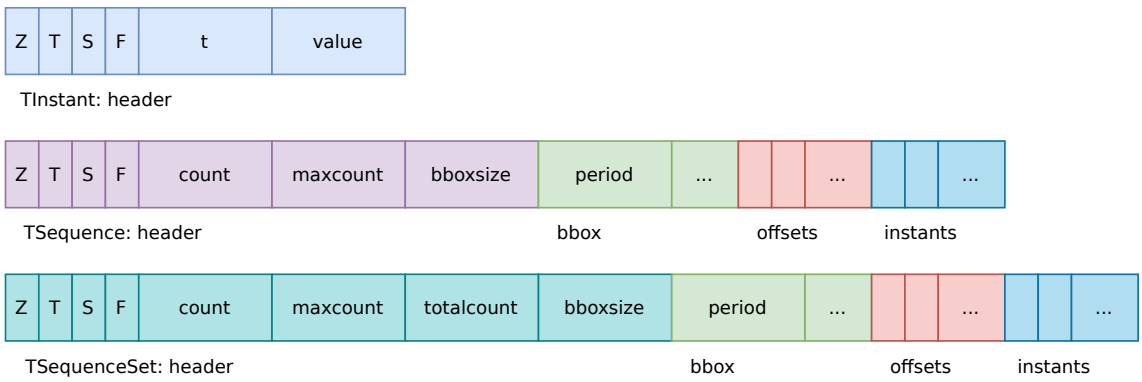

Figure 4.2: MEOS Temporal C structure implementation. [43]

In order to make inheritance work, i.e., to be able to work with a Periodic as if it was a Temporal, the attribute order of the parent Temporal object cannot change within the Periodic object. Otherwise, the program would end up accessing and overwriting memory locations that it is not supposed to, resulting in data corruption.

Moreover, disabling pointer casting would not solve the problem either. Defining the attributes after the variable part will require regularly moving the memory location of these attributes whenever the instant array changes.

Additionally, as neither bounding boxes, offsets, nor instants are included in the `TSequence` structure definition, we cannot simply define Periodic attributes after the variable-length data.

```
typedef struct
{
  int32 vl_len_;
  uint8 temptype; // Temporal type
  uint8 subtype;  // Temporal subtype
  int16 flags;    // Flags
  int32 count;    // Number of TInstant elements
  int32 maxcount; // Maximum number of TInstant elements
  int16 bboxsize;
  char padding[6];
  Span period;
  /* variable-length data follows */
} TSequence;
```

The memory location of these is defined dynamically during sequence instantiation. To emphasize, instants can be accessed purely by "manually" choosing the corresponding memory address; a similar `#define` approach would be required for each Periodic attribute.

```
#define TSEQUENCE_INST_N(seq, index)
  ((const TInstant *)((
    (char *) &((seq)->period)) +
    (seq)->bboxsize +
    (sizeof(size_t) * (seq)->maxcount) +
    (TSEQUENCE_OFFSETS_PTR(seq))
  [index]))
```

Briefly, starting with the memory location of the actual sequence, it adds the memory sizes of different attributes, in order to compute the actual memory position of a single instant.

New Subtype

Instead of appending periodic attributes to existing subtypes, another possible implementation approach attempted to directly add a periodic subtype to the

Temporal class. Similarly to other subtypes, `TPeriodic`[8] would act as a container object, storing instances of `TSequenceSet`.

However, the main drawback of this approach is that it requires refactoring all current Temporal functions in order to take the periodic subtype into account. To demonstrate, the `length` method splits conditionally depending on the subtype of the received temporal object, taking into account the implementation approach and algorithms that vary for each subtype. Similarly, a periodic branching would need to be added.

```
1  double
2  tpoint_length(const Temporal *temp)
3  {
4    if (temp->subtype == TPERIODIC)
5      return tperiodic_length(TPeriodic * temp);
6    else if (! MEOS_FLAGS_LINEAR_INTERP(temp->flags))
7      return 0.0;
8    else if (temp->subtype == TSEQUENCE)
9      return tpointseq_length((TSequence *) temp);
10   else /* TSEQUENCESET */
11     return tpointseqset_length((TSequenceSet *) temp);
12 }
```

Furthermore, a more drastic approach would involve completely rewriting the `Temporal` data type in order to support periodicity by default. As a matter of fact, a temporal sequence is a special case of a periodic sequence where the total number of repetitions is equal to zero.

Nevertheless, the aim of our periodic functionality is to add a side feature to the MobilityDB framework, which might be useful for certain scenarios involving repetitive information. The previously cited implementation attempts would require refactoring the core of MobilityDB. This is not only out of scope for this master's thesis, but especially it would also complicate future development because periodic behaviour would always need to be considered.

On the other hand, the periodic type relies on the implementation of *time*, and it is not as immediate as simply adding a new `temptype` (integer, float, point, ...). In fact, modifying the time behaviour might affect the behaviour and development of all current and possibly future `temptypes`. Yet, splitting the current codebase in

[8] We provide this as a concept; in practice, it might be better to split it into multiple structures such as `PSequence`, `PSequenceSet`, ...

half is neither an option, as (i) too much duplicate code would be made, and (ii) it still needs to work and be made as a MobilityDB's functionality.

Although many other implementation approaches exist, they will not be further discussed since we have addressed the principal issues above.

4.3.2 Chosen Implementation Approach

The approach chosen for the scope of this work focuses on a straightforward implementation approach that can be split into two parts: relative temporal, and periodic information.

Temporal Part

On one hand, we have the Temporal part which implements the relative sequence. The major additions to the Temporal structure are in the input and output methods and concern taking into account different textual representation that make the manipulation of sequences relative to 2000-01-01 more intuitive.

The representation is stored using the `int16 flags` attribute of `Temporal`, similarly to how the interpolation type is stored. The simple idea behind flags in programming is to make use of the binary representation of numbers and treat them as boolean arrays. Each bit of `int16` is treated as an individual boolean value, and bitwise operations can be used to easily set or clear the state of each bit.

```
1 #define MEOS_FLAG_PERIODIC 0x0700
2 #define MEOS_FLAGS_GET_PERIODIC(flags)
3     (((flags) & MEOS_FLAG_PERIODIC) >> 8)
4 #define MEOS_FLAGS_SET_PERIODIC(flags, value)
5     ((flags) = (((flags) & ~MEOS_FLAG_PERIODIC) | ((value & 0x07) << 8)))
```

We thus define the following enumeration to represent possible textual representations of the relative sequence.

```
1 typedef enum
2 {
3   P_NONE     = 0,  // Temporal (not periodic)
4   P_DEFAULT  = 1,  // #2000-01-01 00:00:00
5   P_DAY      = 2,  // #00:00:00
6   P_WEEK     = 3,  // #Monday 00:00:00
7   P_INTERVAL = 4,  // #0 days
8 } perType;
```

Month and year representations are omitted due to numerous drawbacks previously discussed (cf. 4.1.1).

In order to make these input and output textual representations work with MobilityDB, and MEOS in particular, date formatting functions were borrowed and adapted from PostgreSQL[9] including `TO_CHAR()` and `TO_TIMESTAMP()`. Using a series of template patterns, they allow easy formatting of text input and output of timestamps. For instance, the following format allows us to parse and output the weekly style:

```
FMDay HH24:MI:SS.US
-- FMDay: capitalized day name
-- HH24: hour of day (00-23)
-- MI: minute (00-59)
-- SS: second (00-59)
-- US: microsecond (000000-999999)
```

Regarding user input, a `Periodic` keyword might be provided in order to specify the desired `perType`.

```
SELECT tfloat('Periodic=Week; Interp=Step;
  [1#Friday 08:00:00, 2#Saturday 09:00:00, 3#Sunday 10:00:00]');
```

Periodic Part

On the other hand, we have decided to keep periodic attributes (cf. 4.2) separate from the Temporal data type implementation.

The base idea is to create an additional data type we call `PMode`, which acts as an attribute collection, and its purpose is to simply group periodic information into a single structure. In other words, it allows reusing the same configuration of attributes for multiple sequences or returning multiple periodic attributes at once in functions. In other cases, these attributes can be provided separately as arguments depending on function requirements.

Regarding functions, we choose to keep Temporal and Periodic functions separate. For existing MobilityDB functions that can utilize periodic behaviour, we define equivalent signatures with a *periodic_* prefix and required periodic attributes.

```
double tpoint_length(const Temporal *temp);
```

```
double periodic_point_length(const Temporal *temp, const PMode *pmode);
```

In contrast to conditionally splitting the implementation (cf. 4.3.1) and requiring all functions to support periodicity, this approach allows progressively implementing

[9] https://www.postgresql.org/docs/current/functions-formatting.html

new periodic functions depending on the application's needs. In case a periodic version of a function is not (yet) implemented, it can simply be transformed into a Temporal by the anchoring process, and then given to any currently existing MobilityDB function. Being able to use the same function signature for both temporal and periodic sequences is left as possible future work, as the required refactoring is out of scope of this master's thesis.

Moreover, as the relative sequence is implemented as and works like a simple Temporal, all current MobilityDB functions can be applied on relative sequences. This is useful as repetition is not always required for periodic sequences. For instance, a function returning the spatial 2D trajectory is not affected by repetitions or by time in general.

Additionally, whenever a single data structure is required for both the Temporal and Periodic parts, a composite structure could be used to temporarily encapsulate them together.

```
1 typedef struct {
2   Temporal *temp;
3   PMode *pmode;
4 }
```

However, it would simply act as a middleman between database and functions, since storing pointer structures is tricky in the context of database systems. In other cases, data would need to be serialized beforehand.

4.4 Other Concepts

Below, we discuss additional topics related to periodicity that, although not all have been implemented, remain important to address.

4.4.1 Nested Repetitions

As previously discussed with the concept of periodic tree [7] (cf. 3.2.1), nested repetitions allow creating more complex movements than a single period trajectory, using a hierarchy of relative sequences combined with repetition nodes. A simple example is transport vehicle making the same trip 10 times a day and only during weekends: this might involve an hourly repetition period, a weekly repetition period, and some no-movement sequences that fill waiting gaps in between.

Although we have not implemented the periodic tree during our work, a similar structure could possibly be built in the future on top of the ideas we have developed. As a reminder, the tree data structure contains several nodes including a periodic node P and a movement link node L. Similarly, a repetition hierarchy could be constructed in MobilityDB by replacing these nodes with PMode and (relative) Temporal data structures respectively. The former would store either the number of repetitions or the period of the below nodes, while the temporal sequences would construct the complete movement by reading the tree in a depth-first manner.

However, note that constructing such hierarchy would also require the development of specific operations to handle periodicity. Whereas an anchoring operation is immediate, functions such as `periodicValueAtTimestamp` (cf. 4.5.1) would require to actually traverse the tree rather than returning the result of a simple modulo operation.

Regarding the tree construction, its algorithm assumes input sequences are precise and regular, which otherwise would require careful pre-processing.

Moreover, legible input and output textual representations would need to be designed in order to handle sequences in SQL. An initial idea could involve parenthesizing temporal sequences with repetition-related keywords. However, without a concise and proper standard, these could become difficult to read as the tree depth increases.

Additionally, such a tree structure might be difficult to modify, especially regarding the repetition span, as changing the duration of sub-movements or their number of repetitions might produce unexpected time intervals. We will illustrate this with the example from Figure 3.1. The `C(R10 + L7)` node describes a movement of '1 day': `R10` repeats the underlying sequences 10 times, and `L7` fills up the remaining time by 'not moving' during night. Thus, changing the value of, for example, `R10` to `R11`, would require two steps. First, ensure `R11` does not span a period greater than '1 day', as this would alter the movement logic. Second, shorten `L7`'s sequence length accordingly so that the C node spans '1 day' correctly.

4.4.2 Acyclic Periodicity

As discussed before, periodic parametric rectangles (cf. 3.2.2) are defined to enable acyclic periodic movement, which mixes periodic with linear movements. Common examples include the coordinates of a pendulum (periodic) on a uniformly moving boat (linear) or a simplified satellite movement joined with Earth's rotation.

Since acyclic movement is defined as $g(t) + h(t)$ where $g(t)$ is a cyclic periodic movement and $h(t)$ a linear movement, the simplest approach is to respectively

add an anchored periodic sequence with a temporal sequence. For instance, the periodic sequence might store a relative position of the pendulum regarding the boat, while the temporal contains the actual coordinates of the boat (relative to Earth). Note that this approach requires (i) anchoring and repeating the periodic sequence beforehand, and (ii) having the complete trajectory for the linear movement.

Another possible although specific approach could utilize the number of repetitions to add movement to a periodic sequence. Given a positive integer n representing the n^{th} repetition cycle, we could assume the periodic sequence evolves following $n \cdot g(t)$. This would allow modelling movements whose amplitude or intensity increases with each cycle.

4.4.3 Transparency

We can also investigate whether we can make periodicity transparent, i.e., work with periodic sequences as if they were ordinary temporal sequences.

From the user's perspective, we could consider automatically converting temporal sequences to periodic in order to optimize storage space. However, this would require running repetition detection algorithms in the background. Since these algorithms require regularity and precision of repeating patterns, which real-life trajectories rarely contain, such detection routines are inefficient in practice. Moreover, we cannot allow the simplification of movements, such as using reference spots, in an automatic and transparent context: no information should be lost.

From the system's perspective, we might desire to hide whether a function is given a temporal or periodic sequence. For instance, we would want to compare the `tDistance` between a temporal and a periodic object. If function modifications are allowed, then following previous discussions in 4.3.1, we might either wrap a relative sequence with a PMode into a common structure and give that to functions, and using conditional branching split the internal implementation. Otherwise, the best approach might be to simply decompress the periodic sequence into a temporal, i.e., repeat it in time, and thus simply input two temporal sequences instead. The length of the transformed sequence might also be restricted to the length of the second sequence for additional efficiency.

4.4.4 Exceptions

We have defined periodic movements as movements that repeat in equal time intervals. However, in practice, we might encounter exceptions in that motion but still follow a cyclic movement structure. We could distinguish two types of exceptions:

- **Recurrent exceptions:** these are exceptions that are part of the actual

periodic sequence. For instance, a public transportation service that occurs every day except on weekends. Implementation ideas:

  - **Period array**: instead of following a single period, repetitions might use a (cyclic) array of intervals for more complex patterns.

```
e.g. Mon -> Tue -> Wed -> Thu -> Fri -> Mon
[1 day, 1 day, 1 day, 1 day, 3 days]
```

  - **Occurrence array**: boolean array that divides the period into equidistant segments. Similar to how GTFS defines repetition in *calendar.txt*, which contains a Boolean value for each day of the week specifying whether the service should happen that day.

```
ARRAY[1, 1, 1, 1, 1, 0, 0];
-- period = '1 day'
-- i.e., no service on weekends
```

    Similarly to a cyclic interval array, an anchor function cycles through each element for each pattern occurrence. A true value instantiates the ith occurrence of the pattern, whereas a false one skips it.

  - **Sequence Set**: simply define a sequence set over a greater repetition period. However, it is more prone to duplicate segments.

```
{
  [A#Mon 23:00:00, B#Tue 01:00:00],
  ...,
  [A#Fri 23:00:00, B#Sat 01:00:00]
}
```

- **Anchored exceptions:** these are exceptions related to the anchoring in time of the periodic sequence. For instance, we could take a periodic sequence that repeats every week, except during a holiday period. This can be implemented by defining a span set for each exception span and simply subtracting all set values that overlap with the final anchored sequence.

4.5 Operations

One of the goals of the periodic implementation is to allow users to perform operations without needing to instantiate all the actual repetitions of a relative sequence. Although some algorithms may benefit from periodic properties of sequences, others may work fine without them. For this reason, we will start by classifying periodic methods into three categories.

Basic Operations These operations do not involve periodicity and require only information contained in the relative sequence. Examples of basic operations include simple accessor functions such as getting the interpolation or the min/max values. We can also add methods such as returning the 2D spatial trajectory of the trip or the distance to a fixed geometry, as their results are not influenced by the number of repetitions.

Extended Operations Extended operations are similar to basic operations but involve making slight modifications to the result to account for periodicity. These are functions where periodic information can be put to use by simply adjusting the output rather than modifying the function's body itself. Examples may include calculating the total travelled distance. In fact, we can compute the base distance purely using the relative sequence, which we then multiply by the number of repetitions in order to obtain the total distance across all considered repetition cycles. In general, these are operations such as:

$$\text{modifier}(\text{operation}(RelSeq)) = \text{operation}(\text{anchor}(RelSeq))$$

Advanced Operations Advanced or fully periodic operations require additional periodic information and involve specially designed algorithms. An example of an advanced operation is finding the temporal distance between moving periodic vehicles. The Temporal `tDistance` function computes the distance for each intersecting instant, as illustrated by the following code snippet:

```
1 SELECT
2   tgeompoint '[Point(0 0)@2000-01-01, Point(1 1)@2000-01-02]'
3     <->
4   tgeompoint '[Point(0 1)@2000-01-02, Point(1 2)@2000-01-03]';
5 -- [1@2000-01-02 00:00:00]
```

Points `(0 0)` and `(1 2)` are ignored as both sequences intersect at `@2000-01-02`.

Regarding the periodic version of such distance function, the best-case scenario is whenever periodic sequences have their periods synchronized. In that case, it is sufficient to compute the distance only vis-à-vis of their relative sequences and then extend the temporality to match the intersection of the validity time spans of both sequences.

When the periods are not equal between the two sequences, the distance must be computed for the whole anchoring span as the alignment of both sequences changes with each repetition. We will illustrate using two relative pattern sequences $seq_1 = A_1B_1$ and $seq_2 = A_2B_2C_2$ with periods $p_1 = 2$ and $p_2 = 3$ respectively. When we repeat these sequences, and then compare the aligned instants one-to-one, we

obtain the following sequence

$$(A_1A_2), (B_1B_2), (A_1C_2), (B_1A_2), (A_1B_2), (B_1C_2), (A_1A_2), \ldots$$

which repeats starting with A_1A_2 with period equal to 6. More generally, we can compute the least common multiple of the two periods to get the lowest common period ($lcm(p_1, p_2) = 6$). The distance computation might be stopped and extended after lcm steps, i.e., whenever this comparison starts to cycle, as the distances will not change in further repetitions. The worst-case scenario is when the cycle never happens within the complete anchor span. In that case the computational cost is equal to or worse than computing the temporal distance with instantiating each occurrence of the periodic sequence.

As a matter of fact, it might not be always possible to optimize each periodic function, as their result might simply require to instantiate all the occurrences beforehand. In such cases, the basic temporal version of the function might be preferred.

4.5.1 Example Functions

Anchor

This is the fundamental function for periodic behaviour. Although traditionally anchoring simply means assigning a timestamp to it, we define it as the function that actually also repeats relative sequences and transforms them into a complete temporal sequence. Traditional anchoring behaviour can be achieved with either MobilityDB `shiftTime` or `periodicAlign` defined below. Different implementations are possible, but in our case it is divided into two simple parts.

First, it anchors the relative sequence in time. Given a timestamp, the relative sequence is simply shifted by the interval obtained by subtracting that timestamp by the *2000 UTC* reference point. As discussed previously (cf. 4.2), since we require a timestamptz span instead of a timestamp, we use the lower span bound instead.

Second, we copy the base part of the cycle, shift it by the `period` multiplied by the current cycle count, and merge it with the initially shifted relative sequence. This process is basically repeated until we reach the upper anchor bound. Additionally, several stop conditions are defined related to bound inclusion and the `strict_pattern` attribute.

PeriodicValueAtTimestamp

Allows retrieving the value of a periodic at any timestamp without instantiating each repetition occurrence of the sequence.

It uses the simple principle that repeated movements can be represented by a function of the form $f(t \bmod p)$ where t is the timestamp given as an argument, p the period, and mod the modulo operation. Thus, getting the value of a periodic sequence at any timestamp comes down to transforming the timestamp such that it fits within the period span and getting the equivalent value of the relative sequence.

This allows us to efficiently query particular values, which would be time-consuming for especially long anchor durations in the case of instantiating each repetition.

```
SELECT periodicValueAtTimestamp(
  tint('Periodic=Day; [1#08:00:00, 2#08:30:00, 2#09:00:00)'), -- pseq
  '[2024-02-01 00:00:00, 3024-02-01 00:00:00]'::tstzspan, -- anchor
  '2 hours'::interval, -- period
  '2024-03-06 10:45:00'::timestamptz -- arg timestamp
);
-- periodicvalueattimestamp|2
-- Time: 0.123 ms
```

PeriodicValueAtInterval

Similar to `periodicValueAtTimestamp`, but using an Interval.

In fact, since we assume Intervals to be relative to a given timestamp, whether it is the *2000 UTC* start point or an anchor timestamp, an interval equivalent might be added to each timestamp-related MobilityDB function involving a relative or a periodic sequence.

PeriodicAlign

Similar to `timeShift` which shifts all instants by an interval, but instead shifts them such that the first instant starts at the given timestamp. The shift interval is simply obtained by subtracting the sequence's first timestamp from the argument timestamp.

```
Temporal *
periodic_align(const Temporal *temp, const Timestamp ts)
{
  TimestampTz start_tstz = temporal_start_timestamptz(temp);
  Interval *diff = minus_timestamptz_timestamptz(ts, start_tstz);
  Temporal* result = temporal_shift_scale_time(temporal, diff, NULL);
  return result;
}
```

Its main purpose is to be used in an aggregation or comparison scenario when we want to compare multiple relative sequences. By default, it shifts the sequence to the reference *2000 UTC* timestamp. Otherwise, it can be used to anchor the relative sequence to a specific timestamp when repetition is not necessary.

PeriodicType and SetPeriodicType

Allow getting and setting the periodic flag which changes the textual representation of a given relative sequence.

PeriodicCompression

Transforms a temporal sequence with repetitions into a periodic sequence. Returns a composite type including a relative sequence without repetition, as well as a PMode that contains extracted repetition information.

However, note that it only performs a naive and simple repetition detection. For a repetition to be detected, the entire sequence must contain repetitions, and all values and timestamps must be relatively equal across cycles. In other words, neither spatial nor temporal imprecision, gaps, etc. are allowed. Thus, in practice, it requires adequate preprocessing as discussed previously in 3.3.

PeriodicTDistance

As explained in the advanced operation example (cf. 4.5), a `tDistance` function between two periodic sequences can be implemented by anchoring both sequences to a common repetition period. The common period is obtained by computing the least common multiple of each sequence's period. Then, the non-periodic `tDistance` can be called to compute the distance between these semi-anchored sequences. Finally, that computed result can be anchored using the initial anchor span, with anchor's repetition period being the previously computed common period.

If both periods are the same, then the non-periodic `tDistance` function can be directly used on the relative sequences.

AnchorArray

An extension of the previously defined anchor function. As explained in the exception section (cf. 4.4.4) and illustrated in (cf. 5.1.7), it takes an additional boolean array of arbitrary size as argument. For each pattern repetition cycle, a true value indicates that the pattern should be instantiated, whereas a false value simply skips it. The array is read cyclically.

As an illustration, one could create a 7-size array with a 1-day period anchor on a Monday. If the first five values were set to true, and the latest two to false, it would repeat the pattern each day except on weekends.

However, it should be noted that using such an *array* approach may require adapting all other functions to also support the array input. For instance, `periodicValueAtTimestamp` may require additionally computing the cycle number modulo the cyclic array size in order to determine if the cycle is instantiated or not. Similarly, `periodicTDistance` would require considering the least common multiple of periods multiplied by the length of their corresponding cyclic arrays.

Chapter 5

Case Study: STIB/MIVB GTFS

This chapter illustrates and explores the use of periodic sequences for GTFS in MobilityDB.

5.1 Importing GTFS

As discussed previously in 2.4, GTFS is a CSV format that stores transport related information. In this chapter, we will focus on GTFS Static, a subset of GTFS related to timetables. This means that trips are stored with periodic behaviour, since they follow a weekly calendar schedule.

5.1.1 Trips and Periodicity

In a single day, each trip of a public transportation route is defined mainly by three parts.

First, a geometric line shape describes the exact spatial path that the vehicle must follow. It is defined for each route, and it is thus shared by all trips belonging to the same route.

Second, each trip has assigned arrival (and sometimes departure) times for each individual stop on its path. By combining each (`trip_id`, `stop_id`, `arrival_time`) tuple, we can construct a temporal trajectory for each trip. By joining the stop location with the previously defined path line shape, we can interpolate intermediate position points between stops, which allows us to create complete MobilityDB trajectories.

Finally, `arrival_time` does not include date information; it only includes time information regardless of the actual day. Date-related information is stored in the `calendar.txt` part of GTFS Static, which is the part that exhibits periodicity. Each trip has assigned service information which contains two important elements. The first is a 7-element boolean array (or Enum) which for each day of the week indicates

whether the trip should operate or not. The second is a Date Span indicating the start and end dates of the service, with bounds included. This means that we can periodically repeat each trip with a weekly period as long as it is contained within the date span bounds.

5.1.2 Data Source

We will work with STIB/MIVB GTFS files, which are openly available at their data portal[1]. Specifically, we are working with GTFS files exported on 2023-03-28 at 23:49 CEST (Europe/Brussels) and containing calendar ranges from 2023-03-20 to 2023-04-16. This data set sample is especially interesting because the local DST change occurred on 2023-03-26 at 02:00, resulting in GTFS trips across two time zones.

The data used in this case study, along with relevant code and scripts, can be accessed via this master's thesis support repository[2].

To import GTFS into MobilityDB, we will first (i) import CSV files into tables, and then (ii) post-process the tables in order to transform them into actual MobilityDB trajectories. To achieve this, we will follow a GTFS import script from the official MobilityDB workshop[3], which we will further modify to take periodic behaviour into account.

Additionally, our examples will mostly follow a trip from STIB's bus 71[4], which for future reference is specified in GTFS as below:

```
route_id,service_id,trip_id,trip_headsign,direction_id,block_id,shape_id
60,250654060,116621908250654060,"DELTA",0,9207704,071b0240
```

Listing 5.1: Single trip GTFS entry of bus line 71.

5.1.3 Basic Import

The first step is to create tables equivalent to the headers of the GTFS CSV files. Since CSV is a simple and commonly used format, we can use the existing PostgreSQL function `COPY FROM`, which allows copying data between tables and files.

```
CREATE TABLE calendar (
  service_id text,
  monday int NOT NULL,
  tuesday int NOT NULL,
  wednesday int NOT NULL,
```

[1] https://data.stib-mivb.brussels/pages/home/
[2] https://github.com/sswirydo/Master-Thesis
[3] https://github.com/MobilityDB/MobilityDB-workshop
[4] https://www.stib-mivb.be/horaires-dienstregeling2.html?_line=71

```
  thursday int NOT NULL,
  friday int NOT NULL,
  saturday int NOT NULL,
  sunday int NOT NULL,
  start_date date NOT NULL,
  end_date date NOT NULL,
  CONSTRAINT calendar_pkey PRIMARY KEY (service_id)
);
```

```
COPY calendar(
  service_id,
  monday, tuesday, wednesday, thursday, friday, saturday, sunday,
  start_date, end_date)
FROM '/home/GTFS/calendar.txt' DELIMITER ',' CSV HEADER;
```

Then, we convert shapes and stop locations previously discussed into PostGIS's internal format in order to generate actual geometries as visualized in Figure 5.1. As a reminder, MobilityDB is also built as a PostGIS extension, whose functions are easily recognizable by their spatial-type *ST_* prefix.

```
INSERT INTO shape_geoms
  SELECT
    shape_id,
    ST_MakeLine(
      array_agg(
        ST_SetSRID(ST_MakePoint(shape_pt_lon, shape_pt_lat), 4326)
        ORDER BY shape_pt_sequence)
      )
  FROM shapes
  GROUP BY shape_id;
```

```
UPDATE stops
SET stop_geom = ST_SetSRID(ST_MakePoint(stop_lon, stop_lat), 4326);
```

5.1.4 MobilityDB Post-Processing

In order to import the calendar part, the official script generates a series between the `start_date` and `end_date` for each service, and then checks if the generated date's day of the week matches the boolean day-of-week array. This decompresses, for each service, the calendar format into arrays with all the individual dates during which the service occurs.

Figure 5.1: Visualization of STIB/MIVB routes and stops.

Since we work with relative sequences, we do not need the service's date span at this point; it can simply be stored alongside for future anchoring. We can simply adapt the query to generate a 1-week relative series (which is GTFS' period of repetition) and extract the day of the week as a date that will later be used as the basis for our relative sequences. As discussed previously, the first day of the week varies depending on local cultures. Here we simply follow STIB's calendar's day-of-the-week order.

```
1  CREATE TABLE periodic_dates AS (
2    SELECT
3      service_id,
4      date_trunc('day', d)::date AS date
5    FROM
6      calendar c,
7      generate_series('2000-01-01'::date, '2000-01-07'::date, '1 day'::
           interval) AS d
8    WHERE (
9      (monday = 1 AND d = '2000-01-01') OR
10     (tuesday = 1 AND d = '2000-01-02') OR
11     (wednesday = 1 AND d = '2000-01-03') OR
12     (thursday = 1 AND d = '2000-01-04') OR
13     (friday = 1 AND d = '2000-01-05') OR
14     (saturday = 1 AND d = '2000-01-06') OR
15     (sunday = 1 AND d = '2000-01-07')
16   ));
```

Afterwards, the script proceeds as follows[5]. We construct a simple sequence of stop geometry points for each trip, using arrival times and the stop order from the Stop Times. Next, we generate intermediate points between stops using Shape Geometry, which provides the geometry linestring for each route. Following, we add date information to the points, ensuring they are specified in the UTC time zone, as MobilityDB works with timestamps that include the time zone.

```
(date + point_arrival_time) AT TIME ZONE 'UTC' AS t
```

Finally, we aggregate these point sequences into actual Temporal sequences, resulting in MobilityDB temporal trips.

This gives us the following trips_mdb table, which contains the `tgeompoint` trajectory for each trip.

```
CREATE TABLE trips_mdb (
  trip_id text NOT NULL,
  direction_id text NOT NULL,
  service_id text NOT NULL,
  route_id text NOT NULL,
  date date NOT NULL,
  trip tgeompoint,
  PRIMARY KEY (trip_id, date)
);
```

At this point, trips_mdb contains only one trip per trip_id; we still need to include *day of the week* service information. To achieve this, we join the dates in trips_mdb with our previously computed periodic_dates table. The replicated `tgeompoint` is then shifted accordingly by the corresponding number of days using MobilityDB's `shiftTime` function.

```
INSERT INTO
  trips_mdb(trip_id, direction_id, service_id, route_id, date, trip)
SELECT
  trip_id,
  direction_id,
  t.service_id,
  route_id,
  d.date,
  shiftTime(trip, make_interval(days => d.date - t.date)) as trip
FROM
  trips_mdb t
```

[5]Details and code are omitted; they are unnecessary for the scope of this paper and do not differ significantly from the official script.

```
12   JOIN periodic_dates d
13     ON t.service_id = d.service_id
14     AND t.date <> d.date;
```

To demonstrate, we can query all the occurrences of a trip of bus 71 that operates from Monday to Friday.

```
1  SELECT date, asText(trip) FROM trips_mdb
2  WHERE trip_id = '116621908250654060' -- bus 71 (route_id 60)
3  ORDER BY date;
```

As expected, we obtain 5 similar trips, one for each service day of the week.

```
1  2000-01-01
2  [ POINT(4.352683 50.849673)@Monday 16:14:00,
3    POINT(4.352794 50.849594)@Monday 16:14:04.593401,
4    ...
5    POINT(4.405075 50.816745)@Monday 16:52:00 ]
6  ...
7  2000-01-05
8  [ POINT(4.352683 50.849673)@Friday 16:14:00,
9    POINT(4.352794 50.849594)@Friday 16:14:04.593401,
10   ...
11   POINT(4.405075 50.816745)@Friday 16:52:00 ]
```

5.1.5 Anchor

Finally, we can anchor the trips to their actual dates, using the corresponding `start_date` and `end_date` of the service and a weekly repetition.

However, there is a small issue related to dates that needs to be addressed beforehand. First, the day-of-the-week enumeration might include days that are not part of the actual service range. For instance, a service might be defined for business days, but the range only cover a single day (Friday):

```
1  service_id, m,t,w,t,f,s,s, start_date, end_date
2  252933004, 1,1,1,1,1,0,0, 20230324, 20230324
```

Despite potentially unnecessarily defined relative trips, the main issue for anchoring is that, although GTFS presents its calendar from Monday to Sunday, the service start_date does not necessarily begin on a Monday. Since we have matched the order of our relative sequences to start on Monday, anchoring to the start_date will yield incorrect results. An intuitive solution might be to move the start date back to the

nearest Monday; however, this may result in instantiating trips outside the service range.

The easiest solution is to cyclically[6] shift the trips themselves before anchoring, such that the first day of the relative sequence corresponds to the specified start_date. As we have kept similar trips as separate TSequences, we can shift them using the following rule. Given i, the i^{th} day of the week of the start_date, we shift all dates before i by $(7 - i)$ and by $(-i)$ otherwise. To demonstrate, we assume the initial sequence (MON, TUE, WED, THU, FRI, SAT, SUN) and the start_date's day of the week being THU. The resulting shifted sequence (FRI, SAT, SUN, MON, TUE, WED, THU) is obtained by shifting the days respectively by $(+4, +4, +4, -3, -3, -3, -3)$[7]. This can be achieved in PostgreSQL using simple CASE expressions and shifting sequences accordingly.

Once the sequences are correctly updated, we can finally anchor them to the service span in order to decompress them into complete temporal trajectories.

```
SELECT anchor(
    trip, --relative_sequence
    span(c.start_date::timestamptz, c.end_date::timestamptz + '1 day'::
        interval), --anchor
    '1 week'::interval, --period
    true --strict_pattern
  ) as anchor_trip
FROM trips_mdb_week_shifted t
INNER JOIN calendar c ON t.service_id = c.service_id
WHERE trip_id = '116621908250654060';
```

Note that since the end_date is included in GTFS and we work with timestamps, we shift the end bound by 1 day. Moreover, since some trips may operate overnight, it might be desired to shift the upper bound arbitrarily by an additional 3 hours. This results in the following temporal sequence set:

```
anchor_trip|
{
  [POINT(4.352683 50.849673)@2023-03-23 16:14:00+01, ...
  POINT(4.405075 50.816745)@2023-03-23 16:52:00+01],
...
  [POINT(4.352683 50.849673)@2023-04-13 17:14:00+02, ...
  POINT(4.405075 50.816745)@2023-04-13 17:52:00+02]
```

[6] If the shifted sequence contains instants with timestamps < 2000-01-01, these need to be shifted positively up by a period.

[7] Here we assume Monday.i = 0. Actual code may require $(dow + 6)\%7$ as PostgreSQL assumes Sunday.i = 0.

```
8 }
```

We can also notice a problem discussed earlier in 4.1.1. Both sequences repeat correctly in equidistant time intervals from a UTC perspective; however, when they switch from CET to CEST on 2023-03-26 due to the "*Europe/Brussels*" time zone, there is a 1-hour difference in the textual representation.

Note that as MobilityDB Temporal currently only supports timestamps with time zones, the trip can only be displayed with 1-hour differences. A workaround is to either alter the database's time zone, or use `setPeriodicType` to set the sequence as relative, which will display it as a timestamp instead.

5.1.6 Visualization

The anchored trips can also be visualized with QGIS (cf. 2.3.5). Using the MOVE plugin, we can directly query our MobilityDB database through QGIS. This allows us to execute the above `anchor` query, which returns a temporal sequence as a QGIS temporal layer. Furthermore, QGIS allows analysing temporal sequences using the Temporal Controller illustrated by Figure 5.2.

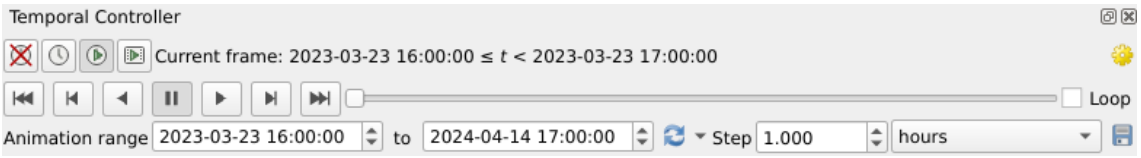

Figure 5.2: QGIS Temporal Controller.

It allows exploring the sequence in time by specifying time frames. In Figure 5.3, we present the complete 1-trip trajectory by choosing a 1-hour time frame such that 2023-03-23 16:00:00 $\leq t <$ 2023-03-23 17:00:00, but it can be adjusted to any time interval.

5.1.7 Daily Repetition

The problem with the current approach is that even though we summarize the movement of trips within a week span, trips are still replicated for several days of the week. Yet we cannot simply repeat the trips each day, i.e, with a '1 day' period as there are days of the week when the service does not take place (e.g., during weekends). Indeed, GTFS describes per week periodic behaviours.

However, as discussed previously (4.4.4), we can still attempt to optimize this by defining custom and more complex repetition behaviours. Strictly speaking, the sequences will not be considered periodic because they will not repeat themselves after exactly a fixed interval. However, in practice, they follow a nested periodic or

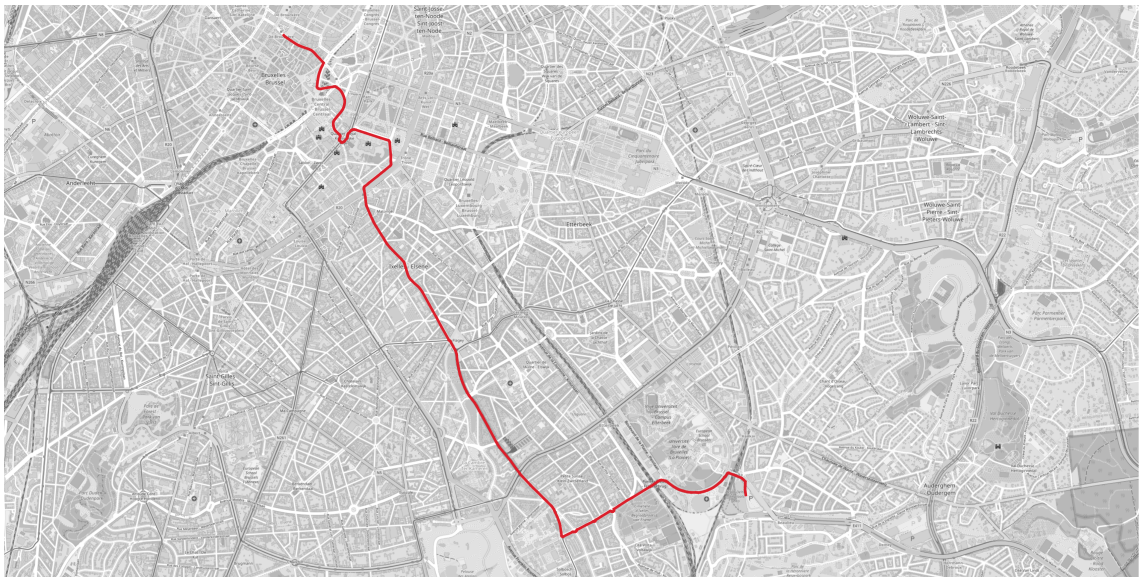

Figure 5.3: Visualization of the Bus 71 trip anchor.

cyclic pattern that can still be utilized. Additionally, losing the periodic property could limit the number of periodic algorithms that can be used on these sequences. We can thus adapt our previously defined `anchor` query to an equivalent `anchor array`.

```
1  SELECT anchor_array(
2    trip, --relative_sequence
3    span(c.start_date::timestamptz, c.end_date::timestamptz + '1 day'::
         interval), --anchor
4    '1 day'::interval, --period
5    true, -- strict_pattern
6    ARRAY[monday, tuesday, wednesday, thursday, friday, saturday, sunday],
         --exception_array
7    (EXTRACT(DOW FROM c.start_date::timestamptz)::int + 6 % 7) --array_shift
8  ) as anchor_trip
9  FROM trips_mdb_day t
10 INNER JOIN calendar c ON t.service_id = c.service_id
11 WHERE trip_id = '116621908250654060';
```

Instead of `trips_mdb_week_shifted`, we use `trips_mdb_day`, which does not replicate trips for separate days of the week. Moreover, the relative start_date problem (cf. 5.1.5) no longer occurs as we only store one trip occurrence, and because we include the shift as the array_shift argument which directly changes the starting index of the array.

5.1.8 Grouping Similar Trips

To further reduce the total number of trips, we might consider grouping similar trips together. Currently, we repeat each trip individually, following the repetition pattern described by the GTFS calendar. However, public transportation routes contain multiple trips that follow the same service and, mostly, exactly the same spatial trajectory. For instance, we can count the number of trips in a single day of bus 71 (see Listing 5.1):

```
SELECT
  AVG(total),
  COUNT(service_id)
FROM
(
  SELECT service_id, count(*) AS total
  FROM trips_mdb_day
  WHERE route_id = '60' AND direction_id = '0'
  GROUP BY route_id, service_id, direction_id
);
```

Which gives us 173 similar trips that share the same service and shape per day on average.

```
avg|173.2
count|5
```

Therefore, we might attempt to group these trips together to further reduce the number of stored trips.

To illustrate, we can consider the following simplified schedule example.

```
TRIP 1: (TRONE 7:20), (BUYL 7:34), (DELTA 7:42)
TRIP 2: (TRONE 15:44), (BUYL 16:00), (DELTA 16:09)
TRIP 3: (TRONE 23:35), (BUYL 23:47), (DELTA 23:54)
```

Which gives the following if we consider these relative to the first instant of each sequence.

```
TRIP 1: (TRONE 0), (BUYL 14min), (DELTA 22min)
TRIP 2: (TRONE 0), (BUYL 16min), (DELTA 25min)
TRIP 3: (TRONE 0), (BUYL 12min), (DELTA 19min)
```

These trips follow exactly the same spatial trajectories; however, they have slight delays relative to each other. A naive solution is to average the intervals in order to obtain an average relative trip.

```
TRIP: (TRONE 0), (BUYL 14min), (DELTA 22min)
```

Which then can be anchored to an arbitrary timestamp such as 13:01.

```
TRIP: (TRONE 13:01), (BUYL 13:15), (DELTA 13:23)
```

Delays

Although this allows drastically reducing the number of trips, the problem with an averaging approach is that we lose delay information between stops. For example, there may be more traffic during the daytime than during the morning or nighttime; these can cause delays which are often estimated by transport companies and directly included in the schedule.

Possible approximations could include computing average delay curves or considering prediction models. However, as these are imprecise and case-dependant, we leave these as possible future work.

Repetition Overlap

Moreover, it should be noted that these average trajectories are relative, but not periodic. In fact, a periodic trajectory repeats itself following regular time intervals. For instance, consider a public transportation line where trips take 30 minutes, with departures every 5 minutes. This means that when repeated with a 5-minute period, the sequence will overlap with itself, resulting in different positions at the same time instants. With some post-processing, it might be possible to distinguish trips by vehicles and organize them to form correct sequences. However, GTFS does not include vehicle information, and in general, such separation is not suitable for scheduling. A better alternative may be to store start timestamps separately in another table, alongside a single average sequence. The sequence can then be anchored for each individual time with a 1-day (or week) repetition period.

Identical Trips

A middle-ground solution to avoid losing delay information while averaging could be to combine identical relative trips. First, we can align our daily trips with `periodic_align`, i.e., make them all start at the same timestamp, as in the simplified example above.

```
WITH aligned_trips AS (
  SELECT
    periodic_align(trip) AS al_trip,
    startTimestamp(trip) AS start_time,
```

```
    ...
  FROM trips_mdb_day
)
```

As Temporal types have an equality operator defined, we can GROUP BY aligned trips to easily see which trips share the same delays, allowing us to potentially group them together.

```
WITH common_trips AS (
  SELECT
    ARRAY(MIN(starttime), MAX(starttime)) AS timerange,
    COUNT(*) AS total
  FROM aligned_trips
  WHERE route_id = '60' AND direction_id = '0'
  GROUP BY route_id, service_id, direction_id, altrip
  HAVING count(*) > 1
)
```

```
SELECT timerange, total FROM common_trips
WHERE service_id = '250656500' ORDER BY total;
```

```
min_start, max_start, total
14:52:00, 18:41:00, 44
12:56:00, 20:30:00, 29
05:30:00, 00:48:00, 15 -- +1 day
11:32:00, 12:50:00, 14
06:29:00, 08:47:00, 12
21:07:00, 22:38:00, 12
13:46:00, 14:46:00, 11
09:41:00, 10:56:00, 11
11:02:00, 11:26:00, 5
09:14:00, 09:32:00, 3
08:56:00, 09:05:00, 2
-- and 6 unique trips
```

The service day is distributed into $11 + 6$ different time splits, reducing the total number of stored trips for that service from 164 to only 17. Note that the used service corresponds to a Saturday. If we replace the service by '250654060', which spans weekdays, we obtain $22 + 7$ different time splits from 214 trips.

Also, we can observe from the above listing that delays are often grouped together depending on the time of the day. However, note that these splits are arbitrarily defined by STIB/MIVB and are not specific to GTFS. Therefore, we cannot consistently guarantee an efficient number of splits.

DTW

Additionally, existing MobilityDB functions such as Dynamic Time Warp[8]

```
1 dynTimeWarpDistance({tnumber, tgeo}, {tnumber, tgeo}) -> float
```

could be used to simplify temporally similar trips up to an arbitrary distance. However, note that such functions require $O(N^2)$ time complexity, with N being the number of trip instants, which might necessitate reducing the MobilityDB trip to stop points only.

Trip Network

Moreover, a potential future work direction would be to consider the approach in [5], where the authors take travel time into account for their frequency-based trip compression. However, it should be noted that this approach was studied in the context of route planning for transportation networks. Thus, trip periodicity is not studied from the perspective of complete line trajectories with multiple stops, but rather at the level of each individual station, as if exploring each node of a network. For instance, node arrivals could be described using two labels: <[8:13,10:13], p=60min, c=2h43min> and <[8:33,10:33], p=60min, c=2h48min>, where both trips have a 60min period but trips starting at 13min have a travel time cost of 2h43min, and those starting at 33min have a travel time cost of 2h48min.

Although this does not correspond to our current approach, it remains interesting; a per-station MobilityDB approach could be studied with tgeompoint trajectories between stations rather than for enclosing complete trajectories.

5.1.9 GTFS Exception Dates

Up to this point, we have not considered an important GTFS file, which is the calendar_dates file (cf. 2.4). The file lists individual exceptional service dates that can be added or removed for each service. This makes the definition of periodic sequences complicated, as it introduces clear irregularities to the repetitions.

Following previous discussions (cf. 4.4.4), a possible workaround is to post-process the anchored periodic sequences. If a service date is removed, the resulting Temporal can be filtered using span overlaps. For instance, MobilityDB defines the function `minusTime(ttype, times)` → `ttype`, which filters temporal sequences based on given time values. Conversely, if a date is added, a 1-day trip can be anchored to that date and merged into the resulting sequence.

[8] https://en.wikipedia.org/wiki/Dynamic_time_warping

However, this cannot be generalized to all periodic functions such as periodic `tDistance`, which specifically optimize the computation based on regular repetitions of sequences. In similar cases, temporal equivalents should be used instead, with periodic sequences anchored and post-processed beforehand.

Otherwise, one might consider mixing periodic and temporal sequences for GTFS, depending on whether the service has exception dates or not. However, this would require adapting queries accordingly to consider both approaches simultaneously.

5.1.10 GTFS Repetition Span

The longer the service [start_date, end_date] interval, the more we can take advantage of periodic behaviour using periodic sequences. Another problem related to STIB/MIVB GTFS is that it covers a relatively short service interval. To illustrate, Table 5.1 shows GTFS trips divided into week buckets based on the length of the service interval. We can observe that not only do all services cover intervals shorter than a month, but recently there are also more non-repeating trips (1 week) than repeating ones. This limits the usability of periodicity for analysis of existing GTFS files. However, we do not have intermediate GTFS data to conclude whether services are becoming more punctual than repetitive, or if this is just a seasonal exception due to the summer school holiday period.

Range	**1W**	**2W**	**3W**	**4W**	**≥5W**	**Total**
March 2023	20.95%	5.99%	7.01%	66.05%	0%	68309
July 2024	70.73%	17.85%	11.4%	0%	0%	123480

Table 5.1: Distribution of trips per service week range.

5.2 Performance Comparison

In this section, we compare how our periodic GTFS approach compares to the classical, non-periodic approach as described in the MobilityDB GTFS workshop.

5.2.1 Test Environment

For reference, all results were computed and measured using the following specifications:

System

- **CPU:** AMD Ryzen 7 5800H 3.20 GHz 8 Core

- **RAM:** 16.0 GB
- **Host OS:** Microsoft Windows 11, version 23H2
- **WSL:** WSL 2.1.4.0 Ubuntu 22.04.4 LTS

PostgreSQL

- MobilityDB 1.1.1
- PostgreSQL 16.3 (Ubuntu 16.3-1.pgdg22.04+1)
- PostGIS 3.4.2

5.2.2 Storage Size

First, we can analyse how the periodic approach impacts the stored data size, presented in Table 5.2. Disk size information was obtained using PostgreSQL `pg_total_relation_size` function[9], which returns the size of the table as well as associated information such as its indexes. The classic relation corresponds to the non-periodic approach, whereas Week, Day, and Group correspond to the approaches described in sections 5.1.4, 5.1.7 and 5.1.8 respectively.

Relation	Classic	Week	Day	Group[10]
Sequence count	506,689	197,401	68,309	13,837
Disk size	4792 MB	1873 MB	1012 MB	142 MB

Table 5.2: Size comparison of periodic approaches.

We can observe a 61% size drop with the regular weekly periodic approach, a 79% for the daily approach, and a remarkable 97% decrease for the additional relative group approach. Although they provide interesting data compression capabilities, note that the daily method limits the set of possible periodic functions, and the group method requires decompression into daily or weekly formats in order to use periodic functions.

Moreover, note that the difference in size is increasingly impacted by the service range specified by the GTFS calendar, as previously discussed in 5.1.10: the longer the service period, the more significant the impact of the periodic approach.

[9] https://www.postgresql.org/docs/current/functions-admin.html

[10] Size does not include the additional startTimes table that is required to differentiate individual relative trips (cf. 5.1.8) and which weights 4064 kB (4 MB).

5.2.3 Execution Time

We will evaluate the time performance of our approach using a query that finds the quickest travel from point A to point B using public transportation at a given timestamp, which is '2023-04-03 12:30:00' in our example.

We start by defining the PostGIS geometries for our start and destination location points, which correspond to two ULB campuses: Solbosch and Plaine.

```
1 WITH args AS (
2   SELECT
3     ST_SetSRID(ST_MakePoint(4.382258393089591, 50.81209088981962), 4326)
        as solbosch,
4     ST_SetSRID(ST_MakePoint(4.39626675936008, 50.81812165343881), 4326) as
         plaine
5 )
```

Then, we identify which routes are within 500 meters of our starting and destination points. Note that this step is particularly important for the non-periodic approach as it drastically reduces the number of evaluated rows.

```
1 near_routes AS (
2   SELECT shape_id
3   FROM shape_geoms
4   WHERE ST_DWithin(shape_geom::geography, (select solbosch from args)::
       geography, 500)
5   AND ST_DWithin(shape_geom::geography, (select plaine from args)::
       geography, 500)
6 )
```

Next, using MobilityDB functions, we identify for each trip the nearest instants to our start and end points.

```
1 temp_near_trips AS (
2   SELECT DISTINCT
3     t.trip_id,
4     nearestApproachInstant(trip::tgeompoint, (select solbosch from args))::
         tgeogpoint AS start_point,
5     nearestApproachInstant(trip::tgeompoint, (select plaine from args))::
         tgeogpoint AS end_point
6   FROM
7     trips_mdb_day t
8     INNER JOIN trips tr ON t.trip_id = tr.trip_id
9   WHERE
```

```
    tr.shape_id IN (SELECT * FROM near_routes)
)
```

Following, we make sure our trip goes in the right direction and restrict the possible times to a 30-minute interval from our starting point using the overlap (&&) operator, in order to reduce the total number of trips for future anchoring.

```
near_trips AS (
  SELECT *
  FROM temp_near_trips
  WHERE getTimestamp(start_point) < getTimestamp(end_point)
    AND start_point && span(
      '2000-01-01 12:30:00 UTC'::timestamptz,
      '2000-01-01 13:30:00 UTC'::timestamptz)
)
```

Then, we anchor the previously obtained trips to the date range specified by GTFS. The `anchor_array` method is used in order to take into account the service days of the week. We also extract the day of the week from the start of service since it is not necessarily a Monday, and use it to shift the starting point of the array.

```
anchored_trips AS (
  SELECT
    t.trip_id, t.route_id, t.service_id, n.start_point, n.end_point,
    anchor_array(
      trip,
      span(c.start_date, c.end_date + '1 day'::interval),
      '1 day'::interval,
      true,
      ARRAY[
        monday, tuesday, wednesday, thursday, friday, saturday, sunday],
      (EXTRACT(DOW FROM c.start_date::timestamptz)::int + 6) % 7
    ) as anchor_seq
  FROM
    trips_mdb_day t
    INNER JOIN near_trips n ON t.trip_id = n.trip_id
    INNER JOIN calendar c ON t.service_id = c.service_id
)
```

Finally, we restrict the anchored values to our initial time interval and order the results by the quickest arrival.

```
SELECT
  trip_id, route_short_name, service_id,
```

```
  getTimestamp(start_point), getTimestamp(end_point)
FROM
  anchored_trips a
  INNER JOIN routes r ON a.route_id = r.route_id
WHERE
  atTime(anchor_seq, span('2023-04-03 12:30:00'::timestamptz, '2023-04-03
      14:00:00'::timestamptz)) IS NOT NULL
ORDER BY getTimestamp(end_point) ASC;
```

For comparison, the non-periodic version of the query basically replaces periodic tables by non-periodic equivalents, and skips the `anchor()` and `atTime()` steps.

The periodic query yields the following trip list:

```
trip_id, route_short_name, service_id, start_ts, end_ts
116621753250654060, 71, 250654060, 12:35:05, 12:39:44
116621256250652000, 71, 250652000, 12:35:05, 12:39:44
116763277251182000, 72, 251182000, 12:36:00, 12:40:48
116621673250654060, 71, 250654060, 12:42:05, 12:46:44
...
```

We can notice that the first two entries are similar in terms of timestamps due to overlapping services for the same group of trips. A possible hypothesis may be that STIB/MIVB sends multiple vehicles at the same time since this is a frequently used bus line that notably joins two ULB campuses. However, as discussed further in 5.1.9, on this particular date, there is a 'remove date' exception entry in the calendar_dates GTFS file for the service 250654060, which may be filtered out.

Notice that we check if the anchored sequence is defined between 12:30 and 14:00 rather than 12:30 and 13:00, which is due to the DST time zone changes. In our case scenario, we anchor to the service start date which, depending on whether it is before or after 2023-03-26, will be either in CET or CEST. As a consequence, trips that are repeated since CET will be represented at 2023-04-03 13:30:00 and those that were anchored starting at CEST at 2023-04-03 12:30:00. Nevertheless, we filter starting points directly at the level of relative sequences, and the above 12:30-14:00 period simply checks if there exists an anchor within the day.

Moreover, we can also notice the same issue in the workshop approach, as trips are created for the earliest date of the service and then replicated and time shifted for each service date, which causes trips that cross time zones to be defined one hour later.

Finally, although in this scenario we intuitively deduce that both times should be at 12:30 CET and 12:30 CEST respectively, it should be noted that GTFS stop

times do not contain any time zone information. Certain edge cases may lead to confusion; for instance, stop times can be unclear for individual trips that cross time zones.

Note that for more pertinent results, the query could be extended by restricting the trip route with stop coordinates, as in practice passengers can only embark at stops points. Recursion could also be added to support transfers, as reaching a certain location often requires using more than one public transport route. However, note that such naive approach might be computational expensive depending on the size of the network. Ideally, route planning algorithms, such as Transfer Patterns [4], which are notably used in Google Maps route planning, should be used.

Performance We can measure the performance of both the periodic and non-periodic versions by using `VACUUM ANALYSE`[11] and `EXPLAIN ANALYSE`[12] PostgreSQL commands. In brief, both of them revolve around PostgreSQL query planning, which determines efficient ways to optimize and execute SQL queries. The former frees space in the database and updates table statistics for optimal execution, whereas the latter details the execution plan chosen by PostgreSQL's execution planner, along with runtime statistics such as query times.

Table 5.3 presents the execution times obtained from 10 query executions.

	Avg	**Std**	**Min**	**Max**
periodic	3705.386	444.884	3454.263	4991.556
classic	15694.405	665.945	15096.372	17572.139
periodic (seqscan=off)	4105.443	230.428	3746.519	4566.602
classic (seqscan=off)	1357.004	53.485	1265.563	1449.269

Table 5.3: Query execution time statistics (in milliseconds) over 10 executions.

From the first two rows, we can observe that the periodic version of the query executed 76% faster than the non-periodic one. Although the non-periodic version requires fewer steps as it skips the anchor, the decreased performance is most certainly due to the higher number of rows to scan.

Thus, we can attempt to disable PostgreSQL sequential scanning with `SET enable_seqscan = off;` to see how it influences the performance. This gives the performance presented in the two latter rows were we can observe a significant duration decrease of the non-periodic approach. There is also a slight time increase for the periodic approach, which is probably due to PostgreSQL choosing to switch

[11] https://www.postgresql.org/docs/current/sql-vacuum.html
[12] https://www.postgresql.org/docs/current/using-explain.html

to Just-in-Time[13] compilation, which in short compiles queries into machine code during runtime for possible speed-up and optimization.

In essence, both approaches performed within acceptable timeframes, with the periodic approach generally showing more stability, despite the additional anchoring step.

[13] https://www.postgresql.org/docs/current/jit.html

Chapter 6

Discussion

This chapter explores the advantages and disadvantages of periodic sequences, evaluates their current state in MobilityDB, and discusses how this project can evolve in the future.

6.1 Relevance of Periodic Sequences

Periodic patterns are present in various domains. Human exhibit periodic behaviours through their daily routines, which involve working and commuting. These repetitions can be analysed in order to optimize routes and schedules and provide insights for numerous applications. In [34], the authors explore real-world applications of periodic pattern mining involving traffic congestion, flight incidents and air pollution analysis. For instance, identifying high congestion regions for improving traffic is not only about reducing travel time and budgets; the authors highlight that developing an efficient transportation system is crucial for saving lives during disasters.

As we have explored in Chapter 5, taking into account periodic properties of such sequences also allows for significant data size compression, especially for long-term repetitions. Periodic compression can be beneficial for storage optimizations, notably involving scheduling and transportation.

6.2 Limitations

Despite the presence of repetitive properties in various applications, periodic sequences present several limitations that can impact their effectiveness in practice.

First, periodic sequences are fragile to imprecision and irregularities. This is especially frequent in GPS sequences which present slight spatial differences and can suffer from irregular sampling due to signal loss or other environmental factors. As explored in Section 3.3, approaches exist to extract periodic behaviours from such

data, but this comes at the cost of data simplification, leading to loss of detail and possibly missing important variations.

Following, real-world data often presents repetitive behaviours but contains slight differences in values or timing, making it non-periodic. As can be seen in 4.4.2, linear movements can complement periodic movements, but those may require additional modelling and effort that might make the use of periodic sequences ineffective.

Moreover, periodic sequences are inflexible and sensitive to exceptions. As seen with GTFS in 5.1.9, the addition or removal of exceptional dates such as holidays requires additional adjustments and increases operational complexity.

Additionally, although periodic sequences are useful for data compression, they are not always the most efficient. Often, results can be deduced directly from relative sequences without needing to consider periodicity, such as trajectories that are not influenced by time (cf. 4.5).

The efficiency of the compression is also dependent on the number of repetitions or the repetition span. As seen in 5.1.10, GTFS data sets, despite showing clear periodic properties, can be limited by the definition of the data set range span, especially if these are limited to a mere one week.

Moreover, handling periodicity with time zones can be particularly challenging. We have discussed in 4.1.1 and encountered additional difficulties throughout 5 due to time zones, but these issues might be more problematic for GTFS using transport companies that operate in areas split by multiple time zones.

In short, maintaining and handling periodic sequences that revolve around complex and real-world data often rely on defining several assumptions that lead to simplifications, inaccuracies and possibly may not hold in practice over the long term and ignore unforeseen events.

6.3 Future Perspectives

There is still room for improvement in our study; below, we suggest several areas that can be further explored.

6.3.1 Locale Support

Locale support refers to an application respecting cultural preferences regarding alphabets, sorting, number formatting, etc., but it is not currently supported by MobilityDB. In PostgreSQL, it affects several procedures including pattern matching (*like*, *similar to*, *regex*), sorting (*order by*), case of text data (*upper*, *lower*, *initcap*), and indexes with *like* and *tochar* functions.

As previously discussed in 4.1.1, it may be used to modify the week-relative

textual representation in order to change the first day of the week according to locale information.

However, according to PostgreSQL documentation, it has an important performance impact. It slows character handling and prevents ordinary indexes from being used by "*LIKE*". For this reason, locale must be used only when it is actually needed.

6.3.2 NPoint

NPoint is an additional MobilityDB type that stands for temporal network point. These add additional constraints to the movement of objects, making them precisely follow specified routes. Instead of inputting the exact geographic position, they allow replacing it with a $[0, 1]$ range specifying the relative position within the route.

Periodicity and GTFS could be further explored and integrated with these network-constrained points to possibly further improve storage efficiency of GTFS trips, as many follow the same route shapes. In fact, network points store geometric information separately, which is stored once for multiple fixed networks.

6.3.3 Scheduling

As we are able to compress periodic scheduling data into compact MobilityDB sequences, it might be interesting to implement functions to make scheduling design easier in MobilityDB.

However, as seen in 5.1.7, public transportation often incorporates delay information directly within the schedules. A first idea might be to allow the creation of delay points or zones that allow slowing down the traffic based on prediction data. However, in practice, a visual and interactive map tool may be more appropriate for this use case.

Moreover, as highlighted by [14], delays are not necessarily due to traffic congestion but also include causes such as vehicle collisions or simple precipitation which are not easy to predict in the long term.

Nevertheless, it may be useful to implement the export of MobilityDB periodic trips directly under the GTFS format.

6.3.4 Periodic Pattern Mining

In 3.3, we have explored how we can extract periodic information from data, but we have not explored how this could be better integrated with MobilityDB. In fact, most approaches involve data mining whose implementation may not be suitable for integration into MobilityDB: (i) this adds complexity that is outside the primary

purpose and goal of this extension, and (ii) data mining often requires case-by-case handling and human-supported plot analysis.

However, it might be interesting to integrate MobilityDB's data with external data mining tools, or to draw guidelines that can be followed for general purposes, not involving too many outliers and complex patterns.

6.3.5 Prediction and Anomalies

Similar to how we can extract periodic sequences from existing data using periodic pattern mining, we could also use these periodic sequences as a basis for predicting future real-world data. Additionally, periodic sequences can help detect scenarios where data diverge significantly from the periodic patterns they are supposed to follow. Below, we briefly explore the research on this topic, which could be investigated in greater detail in the future.

Interesting directions for future prediction in traffic forecasting may be found in [23], where the authors provide an extensive summary on the topic, and in [2], which explores it by extracting sequential patterns from moving object trajectories and building corresponding prediction models.

Anomalies are sudden spikes in data or points that are remarkably different from the rest of the data. An extensive survey of anomalies can be found in [12].

Change detection corresponds to subtler changes over longer periods, or simply, identifying transitions in data i.e., deviation from periodic beyond a certain threshold.

As highlighted in [3], the biggest difficulty in change or anomaly detection lies in identifying the time instant after which the system deviates from its past behaviour. The most common approaches and objectives rely on finding time intervals with similar properties, often based on mean, variance, or other statistical measures.

In a periodic context, the authors of [8] explore land cover change, with applications including tree harvesting, urbanization or agricultural intensification. The foundation of one of their approaches, named the *Recursive Merging algorithm*, relies on seasonal cycles. Indeed, since seasons are expected to be similar from one year to another, a distance or change measure can be defined. The main idea is that similar and consecutive annual seasons can be merged together based on the defined measure, where the final merge will correspond to the change.

Furthermore, the authors of [13] analyse change detection of periodic time series in the context of monitoring. Additionally, [30] explores different examples of periodic time series.

6.3.6 Other Concepts and Domains

In 4.4, we have explored the concepts of nested repetitions and acyclic periodic movements. It may be interesting to further research and analyse these concepts as extensions of the current implementation.

Additionally, as we have analysed periodic sequences with a use case of GTFS, it could be interesting to explore other use cases involving recurrent behaviours using MobilityDB.

Conclusion

Periodic trajectories are present across various domains, including transportation and climate. This master's thesis explored these sequences, and our main contributions are as follows.

We have provided a state-of-the-art of periodic sequences in moving object databases, exploring both how other research represents periodic objects and extracts periodic patterns from data. We have also introduced how these may be utilized for prediction-based analysis.

We have implemented a basis for periodic trajectories within the MobilityDB data management and analysis platform by incorporating periodicity into the core temporal data structure hierarchy.

We have explored how periodicity and MobilityDB can be utilized for simplifying and analysing GTFS Static trajectories.

Finally, we have explored the capabilities and limitations of periodic sequences and concluded by suggesting possible future developments on the subject.

Bibliography

[1] Agrawal, R., and Srikant, R. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)* (San Francisco, CA, USA, 1994), Morgan Kaufmann Publishers Inc., pp. 487–499.

[2] Almuhisen, F., Durand, N., Brenner, L., and Quafafou, M. Sequential patterns for spatio-temporal traffic prediction. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT '21)* (New York, NY, USA, 2022), Association for Computing Machinery, pp. 595–602.

[3] Atluri, G., Karpatne, A., and Kumar, V. Spatio-temporal data mining: A survey of problems and methods. *ACM Comput. Surv. 51*, 4 (Aug. 2018), 1–41.

[4] Bast, H., Carlsson, E., Eigenwillig, A., Geisberger, R., Harrelson, C., Raychev, V., and Viger, F. Fast routing in very large public transportation networks using transfer patterns. In *Algorithms (ESA 2010)* (Berlin, Heidelberg, 2010), Springer, pp. 290–301.

[5] Bast, H., and Storandt, S. Frequency-based search for public transit. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '14)* (New York, NY, USA, 2014), Association for Computing Machinery, pp. 13–22.

[6] Behr, T. *Periodisch bewegte Objekte in Datenbanksystemen.* PhD thesis, FernUniversität in Hagen, Hagen, Germany, 2009.

[7] Behr, T., de Almeida, V. T., and Güting, R. H. Representation of periodic moving objects in databases. In *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems (GIS '06)* (New York, NY, USA, 2006), Association for Computing Machinery, pp. 43–50.

[8] Boriah, S., Mithal, V., Garg, A., Kumar, V., Steinbach, M. S., Potter, C., and Klooster, S. A. A comparative study of algorithms for land cover change. In *Conference on Intelligent Data Understanding (CIDU 2010)* (Mountain View, CA, USA, 2010), NASA Ames Research Center, pp. 175–188.

[9] Cai, M., Keshwani, D., and Revesz, P. Z. Parametric rectangles: A model for querying and animation of spatiotemporal databases. In *Advances in Database Technology (EDBT 2000)* (Berlin, Heidelberg, 2000), Springer, pp. 430–444.

[10] Campello, R. J. G. B., Moulavi, D., and Sander, J. Density-based clustering based on hierarchical density estimates. In *Advances in Knowledge Discovery and Data Mining (PAKDD 2013)* (Berlin, Heidelberg, 2013), Springer, pp. 160–172.

[11] Cao, H., Mamoulis, N., and Cheung, D. W. Discovery of periodic patterns in spatiotemporal sequences. *IEEE Transactions on Knowledge and Data Engineering 19*, 4 (Mar. 2007), 453–467.

[12] Chandola, V., Banerjee, A., and Kumar, V. Anomaly detection: A survey. *ACM Comput. Surv. 41*, 3 (July 2009), 1–58.

[13] Chandola, V., and Vatsavai, R. R. A scalable gaussian process analysis algorithm for biomass monitoring. *Statistical Analysis and Data Mining: The ASA Data Science Journal 4*, 4 (July 2011), 430–445.

[14] Dao, M.-S., Uday Kiran, R., and Zettsu, K. Insights for urban road safety: A new fusion-3dcnn-pfp model to anticipate future congestion from urban sensing data. In *Periodic Pattern Mining : Theory, Algorithms, and Applications*, R. U. Kiran, P. Fournier-Viger, J. M. Luna, J. C.-W. Lin, and A. Mondal, Eds. Springer, Singapore, 2021, pp. 237–263.

[15] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)* (Washington, DC, USA, 1996), AAAI Press, pp. 226–231.

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

[17] Guting, R., Almeida, V., Ansorge, D., Behr, T., Ding, Z., Hose, T., Hoffmann, F., Spiekermann, M., and Telle, U. Secondo: An extensible dbms platform for research prototyping and teaching. In *21st International Conference on Data Engineering (ICDE'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 1115–1116.

[18] Han, J., Dong, G., and Yin, Y. Efficient mining of partial periodic patterns in time series database. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)* (Washington, DC, USA, 1999), IEEE Computer Society, pp. 106–115.

[19] Jindal, T., Giridhar, P., Tang, L.-A., Li, J., and Han, J. Spatiotemporal periodical pattern mining in traffic data. In *Proceedings of the 2nd ACM SIGKDD International Workshop on Urban Computing (UrbComp '13)* (New York, NY, USA, 2013), Association for Computing Machinery, pp. 1–8.

[20] Kasperovics, R., Bohlen, M. H., and Gamper, J. Representing public transport schedules as repeating trips. In *2008 15th International Symposium on Temporal Representation and Reasoning (TIME '08)* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 54–58.

[21] Kasperovics, R., Böhlen, M. H., and Gamper, J. On the efficient construction of multislices from recurrences. In *Scientific and Statistical Database Management (SSDBM 2010)* (Berlin, Heidelberg, 2010), Springer, pp. 42–59.

[22] Kullback, S., and Leibler, R. A. On information and sufficiency. *The Annals of Mathematical Statistics 22*, 1 (Mar. 1951), 79–86.

[23] Lana, I., Del Ser, J., Velez, M., and Vlahogianni, E. I. Road traffic forecasting: Recent advances and new challenges. *IEEE Intelligent Transportation Systems Magazine 10*, 2 (Apr. 2018), 93–109.

[24] Lee, J.-G., Han, J., and Whang, K.-Y. Trajectory clustering: a partition-and-group framework. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)* (New York, NY, USA, 2007), Association for Computing Machinery, pp. 593–604.

[25] Li, Z. Spatiotemporal pattern mining: Algorithms and applications. In *Frequent Pattern Mining*, C. C. Aggarwal and J. Han, Eds. Springer, Cham, 2014, pp. 283–306.

[26] Li, Z., Ding, B., Han, J., Kays, R., and Nye, P. Mining periodic behaviors for moving objects. In *Proceedings of the 16th ACM SIGKDD International*

Conference on Knowledge Discovery and Data Mining (KDD '10) (New York, NY, USA, 2010), Association for Computing Machinery, pp. 1099–1108.

[27] Li, Z., and Han, J. Mining periodicity from dynamic and incomplete spatiotemporal data. In *Data Mining and Knowledge Discovery for Big Data: Methodologies, Challenge and Opportunities*, W. W. Chu, Ed. Springer, Berlin, Heidelberg, 2014, pp. 41–81.

[28] Mamoulis, N., Cao, H., Kollios, G., Hadjieleftheriou, M., Tao, Y., and Cheung, D. W. Mining, indexing, and querying historical spatiotemporal data. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '04)* (New York, NY, USA, 2004), Association for Computing Machinery, pp. 236–245.

[29] Revesz, P., and Cai, M. Efficient querying and animation of periodic spatiotemporal databases. *Annals of Mathematics and Artificial Intelligence 36*, 4 (Dec. 2002), 437–457.

[30] Seymour, L. An overview of periodic time series with examples. *IFAC Proceedings Volumes 34*, 12 (Aug. 2001), 61–66.

[31] Sirisha, G., Mogalla, S., and Raju, G. P. Periodic pattern mining - algorithms and applications. *Global Journal of Computer Science and Technology 13*, C13 (July 2013), 19–28.

[32] Terenziani, P. Symbolic user-defined periodicity in temporal relational databases. *IEEE Transactions on Knowledge and Data Engineering 15*, 2 (Mar. 2003), 489–509.

[33] Tuzhilin, A., and Clifford, J. On periodicity in temporal databases. *Information Systems 20*, 8 (Jan. 1995), 619–639.

[34] Uday Kiran, R., Toyoda, M., and Zettsu, K. Real-world applications of periodic patterns. In *Periodic Pattern Mining : Theory, Algorithms, and Applications*, R. U. Kiran, P. Fournier-Viger, J. M. Luna, J. C.-W. Lin, and A. Mondal, Eds. Springer, Singapore, 2021, pp. 229–235.

[35] Upadhyay, P., Pandey, M. K., and Kohli, N. Mining periodic patterns from spatio-temporal trajectories using FGO-based artificial neural network optimization model. *Neural Computing and Applications 34*, 6 (Mar. 2022), 4413–4424.

[36] Vlachos, M., Yu, P., and Castelli, V. On periodicity detection and structural periodic similarity. In *Proceedings of the 2005 SIAM International*

Conference on Data Mining (SDM 2005) (Philadelphia, PA, USA, 2005), Society for Industrial and Applied Mathematics, pp. 449–460.

[37] Wang, S., Cao, J., and Yu, P. S. Deep learning for spatio-temporal data mining: A survey. *IEEE Transactions on Knowledge and Data Engineering 34*, 8 (Aug. 2022), 3681–3700.

[38] Zhang, D. *Periodic pattern mining from spatio-temporal trajectory data*. PhD thesis, James Cook University, Townsville, Australia, 2018.

[39] Zhang, D., Lee, K., and Lee, I. Hierarchical trajectory clustering for spatio-temporal periodic pattern mining. *Expert Systems with Applications 92*, 1 (Feb. 2018), 1–11.

[40] Zhang, D., Lee, K., and Lee, I. Mining hierarchical semantic periodic patterns from GPS-collected spatio-temporal trajectories. *Expert Systems with Applications 122*, 1 (May 2019), 85–101.

[41] Zhang, D., Lee, K., and Lee, I. Semantic periodic pattern mining from spatio-temporal trajectories. *Information Sciences 502*, 1 (Oct. 2019), 164–189.

[42] Zheng, Y. Trajectory data mining: An overview. *ACM Trans. Intell. Syst. Technol. 6*, 3 (May 2015), 1–41.

[43] Zimányi, E., Sakr, M., and Lesuisse, A. MobilityDB: A mobility database based on PostgreSQL and PostGIS. *ACM Trans. Database Syst. 45*, 4 (Dec. 2020), 1–42.