Master's Thesis

Integrating Symbolic Trajectories Into MobilityDB

Student :

Leila Bourouf

Director :

Esteban Zimanyi

2 June 2026

Université libre de Bruxelles

Abstract

Modern applications rely on large volumes of time-varying data that include not only numeric measurements but also symbolic and contextual information. Such data are common in domains such as mobility analysis and Internet of Things systems, where objects change state over time and each state may contain multiple attributes.

MobilityDB provides native support for temporal and spatio-temporal data but restricts symbolic trajectories to atomic symbolic values. This limitation prevents the direct representation of structured symbolic states, such as multi-attribute sensor observations or contextual activity descriptions, as first-class temporal objects.

This thesis proposes an extension of MobilityDB that supports structured symbolic trajectories by using PostgreSQL's `jsonb` type as the value domain of a new temporal type, `tjsonb`. This approach preserves MobilityDB's temporal model and operators while enabling symbolic states with internal structure and heterogeneous attributes.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Professor Esteban Zimányi, for his guidance, advice, and valuable feedback throughout this thesis. His expertise and insights were essential to the development of this work.

I would also like to thank the MobilityDB community for providing an open and collaborative environment that made this research possible. Their work on MobilityDB and temporal database systems provided the foundation for this thesis.

Finally, I would like to thank my family and friends for their encouragement and understanding throughout my studies.

Contents

Chapter 1

Introduction

1.1 Context and Motivation

The rapid growth of mobility data generated by GPS devices, embedded sensors, IoT systems, and context-aware applications has increased interest in the management and analysis of time-varying data. To handle such data, specialized database systems have been developed for moving objects. These systems primarily focus on geometric trajectory representations, typically modeling movement as a function from time to space. While this approach is effective for traditional spatio-temporal queries, it provides limited support for representing the semantic and contextual aspects of movement that are increasingly important in modern applications.

To address these limitations, previous research introduced the concept of symbolic trajectories [16]. In this model, trajectories are represented as time-dependent functions that map time to values from a categorical domain rather than to spatial locations. These values may represent activities, transportation modes, symbolic locations, or system states, enabling higher-level analyses of behavior and temporal patterns that are difficult to capture using purely geometric representations.

Despite their conceptual advantages, symbolic trajectories remain only partially supported in practical database systems. Most relational and spatio-temporal databases are primarily designed for numeric or geometric temporal data and provide limited native support for temporal symbolic values. Consequently, symbolic trajectory data is often managed using alternative approaches, such as relational decompositions or application-level processing, which increase complexity and may reduce performance and optimization opportunities.

MobilityDB is a robust spatio-temporal database system built as an extension of PostgreSQL [22]. It provides native temporal data types, operators, and indexing mechanisms for managing time-varying data [29]. Its temporal framework and close integration with PostgreSQL make it particularly well suited for mobility-related applications. However, symbolic trajectories are currently supported only when symbolic values belong to atomic domains. This limitation prevents the direct representation of structured, heterogeneous, or multi-attribute symbolic states and restricts the expressive power of the system.

At the same time, many data-intensive applications rely on semi-structured data to represent complex contextual information. Within PostgreSQL, the `jsonb` data type is widely used for storing heterogeneous and structured data because it provides efficient storage, indexing, and querying capabilities. As a result, `jsonb` represents a natural candidate for supporting structured symbolic trajectories while remaining fully aligned

with established relational database technologies.

Against this background, this thesis is motivated by the need to bridge the gap between expressive symbolic trajectory models and their effective support within practical database systems. Specifically, it aims to extend the temporal capabilities of MobilityDB to support symbolic trajectories with structured values, enabling richer representations and queries while preserving native temporal semantics and performance.

1.2 Problem Statement

Symbolic trajectories provide a natural and well-established abstraction for representing the temporal evolution of semantic states, such as activities, symbolic locations, or operational conditions. Within MobilityDB, however, the temporal framework is currently limited to atomic value domains. As a result, symbolic trajectories are restricted to simple labels, preventing the direct representation of structured or heterogeneous symbolic states as native temporal objects.

In the absence of native support, symbolic trajectory data is typically managed through alternative design approaches, such as concatenating multiple attributes into a single atomic value, maintaining separate temporal trajectories for individual attributes, or performing symbolic processing at the application level. Although functional, these approaches introduce additional schema complexity, limit direct querying of symbolic attributes, and reduce the effectiveness of built-in temporal operators and indexing mechanisms.

This situation creates a mismatch between application requirements and system capabilities. Applications that rely on rich symbolic trajectory representations cannot naturally express their data models using the temporal infrastructure currently provided by MobilityDB. In particular, the system lacks a unified mechanism for representing and querying symbolic trajectories whose values are structured, multi-attribute, or heterogeneous over time.

The problem addressed in this thesis is therefore the absence of native support for structured symbolic trajectories within MobilityDB. The objective is to enable symbolic trajectories with complex value domains while preserving the system's temporal semantics, query expressiveness, and performance characteristics.

1.3 Research Objectives

The main objective of this thesis is to extend MobilityDB with native support for structured symbolic trajectories.

To achieve this goal, the thesis pursues the following specific objectives:

- Analyze the limitations of the existing support for symbolic trajectories in MobilityDB with respect to expressiveness, query capabilities, and integration with the temporal data model.
- Design a temporal representation for symbolic trajectories with structured values based on PostgreSQL's `jsonb` type that is consistent with MobilityDB's temporal framework.

- Integrate the proposed representation into MobilityDB, enabling structured symbolic trajectories to be treated as first-class temporal objects.
- Evaluate the proposed approach using real-world use cases and benchmarks against existing relational alternatives, focusing on query formulation, performance, storage requirements, and indexing behavior.

1.4 Research Questions

This thesis investigates how symbolic trajectories with structured and heterogeneous values can be supported as first-class temporal objects within a practical spatio-temporal database system. Although symbolic trajectories have received attention in the literature, their integration into widely adopted database systems remains limited, particularly when symbolic states are complex or multi-attribute.

The research is guided by the following questions:

- **RQ1.** What limitations do the existing temporal data types in MobilityDB present when representing symbolic trajectories with structured or heterogeneous values?
- **RQ2.** How can symbolic trajectories with structured values be represented within MobilityDB while preserving its temporal semantics and query model?
- **RQ3.** To what extent does the `tjsonb` model simplify the formulation of queries for structured symbolic trajectories compared to a relational representation based on timestamped `jsonb` data?
- **RQ4.** What are the system-level implications of the `tjsonb` representation in terms of query performance, indexing behavior, and storage usage compared to a relational approach?

Together, these research questions define the scope of the thesis and guide its contributions.

1.5 Contributions

This thesis makes the following main contributions:

- It identifies the limitations of the existing symbolic trajectory support in MobilityDB, which restricts symbolic values to atomic domains and prevents the native representation of structured symbolic states.
- It proposes a new temporal type, `tjsonb`, that represents symbolic trajectories whose values are structured JSON documents while preserving the temporal semantics of MobilityDB.
- It integrates `tjsonb` into MobilityDB and MEOS by reusing the existing temporal architecture, operators, and indexing mechanisms without introducing new temporal abstractions.

- It extends MobilityDB's temporal framework to support the storage, querying, and processing of structured symbolic trajectories using the same principles and abstractions employed by existing temporal types.
- It evaluates the proposed approach using a real-world IoT telemetry dataset and assesses its practicality with respect to query formulation, storage requirements, indexing behavior, and performance compared to a relational model based on timestamped `jsonb` values.

1.6 Thesis Structure

This thesis consists of five chapters.

Chapter 1 introduces the context and motivation of the work. It presents the problem addressed by the thesis, defines the research objectives and research questions, and summarizes the main contributions.

Chapter 2 presents the background and related work relevant to this thesis. It reviews temporal data models, geometric, semantic, and symbolic trajectories, and existing database support for temporal and symbolic data. It also introduces the technical foundations used in this work, including MobilityDB and PostgreSQL support for JSON and JSONB.

Chapter 3 presents the proposed approach for representing structured symbolic trajectories in MobilityDB. It introduces the `tjsonb` type and describes its integration into MobilityDB and MEOS, including its representation, operators, query processing, and indexing support.

Chapter 4 evaluates the proposed approach using an IoT telemetry benchmark. The evaluation examines query formulation, ingestion cost, query performance, indexing behavior, and storage characteristics compared to a relational model based on timestamped `jsonb` values.

Finally, Chapter 5 summarizes the main results, discusses the limitations of the work, and outlines directions for future research.

Chapter 2

Background and Related Work

2.1 Research Background

2.1.1 Time-Varying Data in Database Systems

Many facts in the real world change over time, such as employee roles, salaries, and department memberships. Traditional relational database systems were designed primarily to represent the current state of data and do not provide native support for temporal semantics, making the representation and analysis of historical information more difficult.

Early database applications often relied on timestamp attributes that were managed at the application level rather than by the database system itself [25]. In this setting, the database has no explicit notion of how data evolves over time. As a result, reconstructing past states requires complex and fragile logic, and time-dependent queries are difficult to formulate and maintain correctly.

Temporal database research addresses these limitations by integrating time directly into the database system. Time becomes part of the data model, query semantics, and supported operations, enabling systematic and declarative handling of time-varying data [6, 25].

2.1.1.1 Temporal Data Models

Temporal data models describe how time is associated with database facts. A fundamental distinction is made between valid time and transaction time [25]. Valid time refers to the period during which a fact is true in the modeled reality, while transaction time refers to the period during which that fact is stored in the database. This distinction allows database systems to separate changes in the real world from changes in the database state, which is essential for auditing, traceability, and historical analysis.

Temporal data models also specify how time itself is represented. Most approaches assume a discrete and ordered time domain and distinguish between time instants and time intervals. Instants correspond to single points in time, whereas intervals correspond to contiguous periods during which values remain unchanged. Because many real-world facts remain stable over periods of time, interval-based representations are widely used. In such representations, a value is stored once together with its validity interval rather than being repeated at each individual time point, which reduces redundancy and makes data evolution explicit [25].

Another important design decision concerns how temporal information is attached to data. In tuple timestamping, time intervals are associated with entire tuples, so any

change to an attribute results in the creation of a new tuple. In contrast, attribute-value timestamping associates temporal information directly with individual attribute values, allowing attributes of the same entity to evolve independently. This flexibility reduces data duplication but increases schema complexity and query complexity, leading to trade-offs in storage cost, expressiveness, and query difficulty [25].

2.1.1.2 Temporal Query Semantics and Operators

When data varies over time, queries must also account for temporal variation. Temporal queries are therefore defined over relations whose contents change over time rather than over a single static relation [25].

A central concept in this setting is snapshot semantics. Under this interpretation, a temporal relation can be viewed as a sequence of conventional relations, each representing the database state at a particular time instant. For any given time instant t, the snapshot consists precisely of those tuples that are valid at t. Queries are evaluated independently on each snapshot, which implies that query results may differ across time instants.

To support such behavior, temporal query languages extend relational algebra with operators that explicitly reason about time. These extensions include temporal selection, temporal joins, and temporal aggregation, all of which operate on time intervals and support temporal relationships such as overlap and containment [6, 25].

The literature further distinguishes between current, sequenced, and non-sequenced queries [25]. Current queries restrict evaluation to the present state. Sequenced queries preserve temporal variation and return results together with their validity intervals. Non-sequenced queries treat time as ordinary data without special temporal semantics.

By embedding temporal semantics directly into the query model, temporal database systems reduce the need for manual timestamp manipulation in application code and ensure a consistent interpretation of time-dependent queries [25].

2.1.1.3 Temporal Data Types

Beyond schema-level temporal models and query semantics, temporal database research introduces temporal data types as first-class database values [5]. These data types represent how individual values change over time by combining time and value into a single database object.

A temporal value can be understood as a mapping from a time domain to a value domain, where the evolution of the value is represented explicitly as part of the database object. This representation allows a single temporal object to capture the complete history of a value while supporting temporal operations such as restriction to a time interval or evaluation at a specific time.

Temporal data types provide a foundation for modern temporal and spatio-temporal database systems by integrating temporal semantics directly into data representation and query processing. Compared with timestamp-based approaches, they enable more systematic handling of time-varying information and support richer temporal operations within the database system itself [5, 6, 25].

2.1.2 Trajectory Data as Temporal Objects

Trajectories constitute an important class of time-varying data, as they describe how the position of an object evolves over time. In spatio-temporal databases, trajectories are

usually modeled as functions from time to spatial positions, which describe how an object moves in geographic space [15]. Depending on the application, the spatial position may represent a point, a line, or a region that changes over time. This view connects trajectory modeling with temporal data models, while space appears as an additional dimension that is handled together with time.

2.1.2.1 Raw and Geometric Trajectories

In their simplest form, trajectories are represented as *raw trajectories*. A raw trajectory is a sequence of spatio-temporal samples, where each sample records the position of an object at a given time instant. Such samples are often represented as tuples of the form (x_i, y_i, t_i) and are commonly produced by positioning technologies such as GPS [15]. For example, a GPS device may record the position of a vehicle every few seconds, resulting in a sequence of points that describe where the vehicle was at each recorded time.

From a modeling perspective, geometric trajectories provide an abstraction over raw samples. Instead of treating each recorded point separately, a geometric trajectory represents movement as a single spatio-temporal object. Formally, a geometric trajectory can be defined as a function

$$\tau : T \to R^2,$$

which maps each time instant in a temporal domain T to a spatial location. In practice, this function is often represented using a discrete sequence of points together with interpolation methods. A common approach uses piecewise linear interpolation, where straight-line segments connect consecutive samples. For instance, two GPS points recorded at different times may be connected by a line segment that approximates the path followed between those times.

Geometric trajectory models support quantitative analysis of movement. Typical operations include computing traveled distances, estimating speed and acceleration, and comparing trajectories using geometric similarity measures [15]. These models focus on the geometric properties of movement and form the basis of many spatio-temporal database systems.

2.1.2.2 Database Support for Geometric Trajectories

Spatio-temporal database systems extend spatial databases with explicit support for moving objects and geometric trajectories. In these systems, trajectories are treated as first-class spatio-temporal objects and are represented using specialized data types together with dedicated query operators and indexing structures [15]. This enables databases to store, query, and process movement data directly without relying on application-level logic.

Database support for geometric trajectories typically includes operations that retrieve the position of an object at a given time, extract a sub-trajectory over a specified time interval, and evaluate spatial relationships between trajectories. For example, a query may ask where a vehicle was at a specific time or whether two trajectories intersect during a given period. These operations rely on numeric and geometric computations defined over spatial coordinates and time values.

To execute such queries efficiently, spatio-temporal databases use indexing structures that combine spatial and temporal information. A common technique uses bounding

boxes that summarize the spatial extent of a trajectory segment over a time interval. These summaries are stored in spatial or spatio-temporal indexing structures, often based on hierarchical bounding-box representations. During query processing, indexes quickly eliminate trajectory segments that cannot satisfy spatial or temporal conditions, thereby reducing the number of exact geometric computations that must be performed [15].

These indexing and query mechanisms are primarily designed for numeric and geometric domains, where ordering, distance, and intersection are naturally defined. As a result, geometry-centric trajectory models are well suited for applications that focus on movement analysis, navigation, and traffic monitoring. At the same time, these assumptions limit their ability to represent and query symbolic or semantic information as native time-varying objects.

2.1.2.3 Limitations of Geometry-Centric Models

Although geometric trajectory models describe movement accurately, they focus primarily on the physical aspects of motion. They encode *where* an object moves and *when* this movement takes place, but they do not represent *what the movement means* in terms of activities, behaviors, or context.

Studies in movement analysis show that trajectories with similar or even identical geometric properties can correspond to very different real-world situations [11]. For example, two individuals may follow nearly the same route between the same locations at similar times, while one is commuting to work and the other is participating in a leisure activity. From a geometric perspective, the trajectories appear similar, but their underlying activities and purposes differ. Conversely, the same activity may produce different geometric trajectories due to factors such as traffic conditions, route preferences, environmental constraints, or differences in data sampling. These examples show that geometric similarity does not imply semantic similarity.

As a result, geometry-centric models lack the expressive power needed to represent symbolic states, activities, or contextual information associated with movement. This limitation motivates the development of higher-level trajectory models that attach symbolic or semantic meaning to temporal segments of a trajectory.

2.1.3 Semantic Trajectories

While geometric trajectory models focus on the physical evolution of position, many applications require an understanding of movement at a higher level of abstraction. In many applications, movement data is not interpreted solely as a continuous stream of coordinates. Instead, users and systems reason in terms of meaningful units such as trips, activities, stops, or phases of movement. Raw and geometric trajectories, even when temporally precise, do not provide these abstractions and therefore remain poorly aligned with application-level reasoning [26].

Moreover, movement data is often irregularly sampled and influenced by factors such as sensor availability, user behavior, and environmental constraints. These factors introduce variability that cannot be resolved at the geometric level alone. Interpreting movement therefore requires associating parts of a trajectory with contextual information, such as places, transportation modes, or application-specific states, which are external to the geometric representation [20].

Semantic trajectory models address these limitations by enriching movement data with information that reflects how trajectories are interpreted and used in real-world

contexts. By explicitly linking trajectory segments to meaningful domain concepts, they enable movement to be segmented, interpreted, and queried in terms that correspond to application semantics rather than solely to spatial and temporal measurements [1].

2.1.3.1 Trajectory Segmentation

Raw trajectories are usually recorded as sequences of spatio-temporal positions that describe how an object moves over time. However, many applications do not treat a trajectory as a single continuous path. Instead, they reason in terms of meaningful parts of movement, such as trips, phases, or episodes. Trajectory segmentation addresses this need by dividing a trajectory into a sequence of sub-trajectories that are relevant for a given application [26].

Formally, a segmented trajectory is obtained by partitioning the temporal extent of a trajectory into contiguous time intervals. Each interval corresponds to a sub-trajectory, often called an *episode*. An episode represents a maximal time interval during which the movement satisfies a given homogeneity criterion. Such criteria are often defined using simple movement properties, such as speed, distance traveled, duration, or periods of stillness [20]. For example, a segment may correspond to a time interval during which the speed remains below a given threshold, indicating a stop.

Segmentation rules are usually defined using heuristic thresholds and depend on the application context. A stop may be defined as a period during which the object does not move beyond a certain distance for a minimum duration, while a move may be defined as a period during which the object travels continuously above a speed threshold. These thresholds are chosen based on how movement is interpreted in a specific domain, such as pedestrian mobility, vehicle tracking, or animal movement analysis [20]. As a result, the same trajectory may be segmented in different ways depending on the application goals.

A well-known and widely used segmentation approach is the *stop-and-move* model. In this model, a trajectory is decomposed into alternating segments of stops and moves. Stops represent periods of semantically meaningful inactivity, while moves represent periods of displacement. Importantly, stops are not defined solely by geometric immobility, but by semantic relevance for the application, such as a significant halt at a location rather than a short incidental pause [26].

2.1.3.2 Semantic Annotation of Trajectories

Semantic annotation extends segmented trajectories by introducing information that captures the meaning of movement over time. While trajectory segmentation organizes movement into temporal episodes based on geometric and temporal criteria, semantic annotation enables these episodes to be interpreted in terms of activities, contexts, or symbolic states rather than only as geometric motion [20, 26].

A semantic trajectory is therefore defined as a segmented trajectory in which each episode is associated with semantic information describing the movement or state during its time interval. This information is commonly represented using symbolic descriptors attached to episodes, so that each episode is characterized not only by its temporal extent but also by its semantic meaning [20]. For example, a trajectory may consist of successive episodes labeled *home*, *commute*, and *work*.

Single-Label Semantic Trajectories Early semantic trajectory models represent the meaning of each episode using a single symbolic label, such as a visited place, an

activity, or a transportation mode. This approach constitutes a significant improvement over purely geometric representations by making movement interpretable at a semantic level [20, 26]. However, it implicitly assumes that the semantics of an episode can be adequately captured by one atomic descriptor.

In many real-world scenarios, this assumption is restrictive. Movement episodes often involve several semantic dimensions at the same time. A stop may be associated with a location, an activity, and specific contextual conditions, while a move may involve a transportation mode, traffic conditions, and external events. Encoding such information into a single label results in a loss of semantic detail and limits how trajectories can be analyzed [4].

This restriction also impacts query expressiveness. Queries that combine multiple semantic conditions, such as identifying episodes that involve a particular activity at a specific type of place after traveling by a certain mode of transport, are difficult to formulate when semantics are stored as opaque, single-valued labels. Since labels lack internal structure, databases cannot easily reason about or combine different semantic dimensions [20].

Multiple-Aspect Trajectories To overcome the limitations of single-label semantics, more general models represent trajectories using multiple semantic aspects. In these models, the context of movement is described by a set of semantic dimensions rather than by one label per episode, allowing richer and more flexible representations of movement [4, 18].

An aspect represents a real-world fact that is relevant for interpreting movement, such as place, activity, transportation mode, environmental condition, or social attribute. Each aspect may have its own structure and may be associated with entire trajectories, individual episodes, or specific time intervals [18]. This allows different applications to capture only the semantic dimensions that are relevant to their domain.

The CONSTAnT model illustrates this approach by defining a set of semantic components such as goals, behaviors, environments, and places within a single conceptual schema [4]. Multiple Aspect Trajectories (MATs) generalize this idea by allowing an open and extensible set of aspects that can vary across applications and over time [18]. This flexibility supports a more faithful representation of real-world mobility, where semantic requirements differ between use cases.

From a data management perspective, modeling multiple aspects introduces additional complexity. In relational database systems, semantic aspects are typically stored in separate relations and linked to trajectory episodes through identifiers and temporal conditions. Queries involving multiple aspects therefore require joins across several relations together with temporal predicates. As the number of aspects increases, query formulation becomes more complex and query execution may incur additional overhead. These challenges motivate representations that can capture multiple semantic dimensions within a unified temporal object.

2.1.3.3 Limits of Semantic Trajectory Models

Semantic trajectory models represent an important step beyond geometric trajectories by attaching meaning to movement segments. They make it possible to describe movement in terms of activities, places, and contexts rather than only in terms of spatial coordinates. However, when considered from a data management and system integration perspective,

existing semantic trajectory models exhibit several fundamental limitations.

A first limitation concerns structural complexity. Semantic trajectory models typically rely on rich conceptual schemas in which trajectories, segments, activities, places, events, and other contextual elements are modeled as distinct but interrelated entities. While this expressiveness is valuable at the modeling level, it leads to fragmented data representations in practice. Semantic trajectories are often mapped to multiple relational tables or graph structures, which makes storage less uniform and increases the complexity of data access and maintenance [20, 26].

A second limitation arises in query formulation and processing. Semantic annotations are usually treated as metadata associated with trajectory segments rather than as first-class temporal values. As a result, queries over semantic trajectories often require multiple joins, navigation of conceptual relations, and explicit handling of temporal conditions. This leads to queries that are difficult to express, hard to maintain, and poorly suited for optimization by database systems [4].

A further limitation concerns the separation between temporal behavior and semantic content. In most semantic trajectory models, temporal evolution is captured implicitly through segmentation, while semantic information is attached externally to segments. This design makes it difficult to treat semantic states as independent time-varying entities. Consequently, generic temporal operations such as restriction, comparison, or combination cannot be applied uniformly to semantic information across different domains [20].

Finally, support for semantic trajectories at the database system level remains limited. Although several research models and prototypes have been proposed, semantic trajectories have only limited support as native database types with well-defined temporal semantics and algebraic operations. Instead, semantic processing is often handled outside the database engine or encoded using application-specific logic, which limits scalability, interoperability, and integration with standard query languages [16].

2.1.4 Symbolic Trajectories

The limitations of semantic trajectory models reveal a clear tension between semantic expressiveness and system-level usability. Rich semantic models capture multiple aspects of movement and context, but their representations often become structurally complex. As a result, they are difficult to manage as uniform temporal objects within database systems. In particular, semantic information is typically distributed across segments, entities, and relationships rather than being represented as a single time-varying symbolic value.

Symbolic trajectory models address this challenge by focusing on the temporal evolution of symbolic states. Instead of representing meaning through interconnected conceptual entities, a symbolic trajectory represents the temporal evolution of symbolic states as a time-dependent value [16]. This abstraction preserves the essential temporal behavior of symbolic change while avoiding the structural complexity of richer semantic models.

From a data management perspective, this abstraction allows symbolic states to be treated as first-class temporal values, similarly to temporal numbers or temporal geometries. When symbolic trajectories are modeled in this way, they support generic temporal operators, efficient indexing, and declarative querying based on uniform algebraic constructs without requiring application-specific navigation of semantic schemas [16].

Symbolic trajectories complement semantic models by providing a system-oriented representation of symbolic evolution that is independent of how semantic meaning is

derived. By separating symbolic temporal behavior from semantic structure, symbolic trajectories offer a practical and scalable basis for querying, analyzing, and managing evolving symbolic states in database systems.

2.1.4.1 Definition and Formal Model

Symbolic trajectories describe how symbolic states evolve over time using a single time-dependent object. Formally, a symbolic trajectory is defined as a function

$$f : T \to \Sigma,$$

where T is a temporal domain and Σ is a symbolic value domain [16]. The symbolic domain represents the set of symbolic values that may occur over time. In existing symbolic trajectory models, these values are typically treated as atomic symbols.

In symbolic trajectories, symbolic states are modeled directly as time-varying values. The model does not prescribe how symbolic meaning is derived or how symbolic values are interpreted. Instead, it focuses on capturing when symbolic states are valid over time. This distinguishes symbolic trajectories from semantic trajectory models, which attach semantic annotations to trajectory segments and rely on external conceptual structures to interpret meaning.

In practice, symbolic trajectories are commonly represented using an interval-based model. The temporal domain is divided into a finite set of disjoint time intervals, where the symbolic state remains unchanged within each interval. A symbolic trajectory can thus be represented as an ordered sequence of pairs

$$\langle([t_1, t_2), s_1), ([t_2, t_3), s_2), \ldots, ([t_n, t_{n+1}), s_n)\rangle,$$

where each interval $[t_i, t_{i+1}) \subseteq T$ is associated with a symbolic value $s_i \in \Sigma$ [7, 16].

This representation corresponds to a piecewise-constant temporal function. Symbolic states persist over time and change only at interval boundaries, which represent state transitions. This temporal behavior matches many applications where symbolic information evolves in discrete steps while remaining stable between changes.

2.1.4.2 Querying and Analysis of Symbolic Trajectories

Since symbolic trajectories represent symbolic states as sequences of values that are valid over time intervals, they support querying and analysis tasks that are difficult to express directly on geometric or semantic trajectory models. One important class of operations is pattern matching, where a query specifies a sequence of symbolic values or constraints and retrieves trajectories, or parts of trajectories, that match this pattern [16].

Pattern matching queries describe symbolic sequences together with temporal conditions. For example, a query may ask for all trajectories where the symbolic state *bus* is followed by *walk* within a given time window. Such a pattern may also allow unspecified symbols between these states. Güting et al. propose a pattern language for symbolic trajectories that supports wildcards, ordering constraints, and temporal restrictions, and that allows retrieval of full trajectories or specific sub-sequences that satisfy the pattern [16].

Many pattern matching approaches rely on automata-based or regular-expression semantics. In this setting, symbolic patterns are translated into automata that accept sequences of symbolic states satisfying the query. This approach makes it possible to

express complex temporal and symbolic constraints in a clear and formal way. To improve performance, indexing techniques can be used to avoid scanning all trajectories. For example, symbolic values may be indexed using tree-based structures combined with temporal information, which helps prune irrelevant candidates during query evaluation [27].

Symbolic trajectories also support similarity search and classification tasks. Instead of comparing precise geometric trajectories, similarity is evaluated over symbolic representations that preserve the order and frequency of symbolic states. Early work represents movement as symbolic strings and applies edit-distance-based measures to compare trajectories efficiently, using symbolic filters to reduce computational cost [7]. Later approaches extended this idea by using weighted or normalized edit distances to compare symbolic movement sequences for clustering and related classification tasks [10]. By operating on compact symbolic representations rather than raw geometric trajectories, these methods reduce the need for costly geometric computations and often improve efficiency in practice.

In addition to retrieval, symbolic trajectories support transformation and abstraction operations. A rewriting operation extracts parts of a trajectory that match a given symbolic pattern and produces a new symbolic trajectory that can be analyzed further [16]. Classification operations group symbolic trajectories into categories based on predefined pattern sets, which supports mining and higher-level analysis tasks.

Existing symbolic trajectory models generally assume that symbolic states belong to atomic symbolic domains. Typical examples include activity labels, transportation modes, symbolic locations, or system states represented as single categorical values. While this assumption simplifies representation and query processing, it limits the ability of symbolic trajectories to capture structured symbolic information composed of multiple attributes. As modern applications increasingly rely on heterogeneous and semi-structured data, this restriction creates a gap between the expressive requirements of symbolic state representation and the capabilities offered by existing symbolic trajectory models.

2.1.4.3 Applications and Use Cases

Symbolic trajectories were introduced to support applications in which the evolution of symbolic states over time is more relevant than precise geometric detail. By abstracting movement or behavior as sequences of time-interval–symbol pairs, symbolic trajectories enable analysis tasks that are difficult or inefficient to perform on geometric or richly structured semantic models [16].

A primary application domain is *human mobility analysis*. In this context, symbolic trajectories are used to represent activities, transportation modes, or visited places as temporally ordered symbolic states. This abstraction supports tasks such as activity pattern mining, routine detection, and similarity analysis across individuals, where the focus lies on comparing symbolic behaviors rather than spatial paths [9].

Symbolic trajectories are also applied in *transportation and traffic analysis*. Here, symbolic values may encode road categories, traffic conditions, or vehicle states over time. Representing such information symbolically enables the detection of recurrent mobility patterns, behavioral classification, and the evaluation of temporal correlations between symbolic states and external factors while avoiding the complexity of full spatio-semantic models [16].

Beyond mobility, symbolic trajectories naturally generalize to *system and behavior modeling*. Any system whose state evolves discretely over time such as communication

networks, industrial processes, or software systems can be represented as a symbolic trajectory. In this setting, symbolic trajectories provide a uniform abstraction for representing logs, operational states, or mode transitions as temporal symbolic values, enabling querying, pattern matching, and behavioral analysis using temporal operators [23].

These applications illustrate that symbolic trajectories are not limited to movement data. Instead, they constitute a general-purpose model for representing and analyzing the temporal evolution of symbolic states across a wide range of domains. At the same time, existing symbolic trajectory models typically assume atomic symbolic values, which limits their ability to represent structured or heterogeneous symbolic states directly. This limitation is particularly relevant for modern applications in which symbolic states often consist of multiple attributes or semi-structured information that evolves over time.

2.1.5 Symbolic Trajectories in Database Models

This section examines how symbolic trajectories are represented in database systems, with a focus on the type of symbolic values that are supported and the degree of integration with temporal query processing. It first discusses atomic symbolic trajectories, which are supported by several research and database systems, and then analyzes the limitations that arise when symbolic states become more complex.

2.1.5.1 Atomic Symbolic Trajectories

Atomic symbolic trajectories represent the simplest form of symbolic trajectory modeling. In this setting, the symbolic value domain consists of atomic symbols, that is, labels without internal structure. Typical examples include transportation modes such as *walk*, *bus*, or *train*, activities such as *home*, *work*, or *shopping*, and system states such as *idle*, *active*, or *faulty* [16]. Each symbol represents a symbolic state that is valid over a specific time interval.

An atomic symbolic trajectory describes how such labels evolve over time. For example, a daily mobility trace may be represented as a sequence in which an object walks, then takes a train, and then walks again, with each behavior valid during a given interval. Similarly, a system log may record transitions such as *idle* followed by *active*. In all cases, symbolic values are treated as opaque identifiers, and the model does not assume or exploit any internal structure within the symbols.

Formally, an atomic symbolic trajectory can be modeled as a function $f : T \to D$, where T is the temporal domain and D is a finite or countable set of atomic symbols. In practice, this function is implemented as a finite sequence of disjoint time intervals, each associated with exactly one symbolic value, resulting in a piecewise-constant temporal representation [16].

Support in Research Systems Research systems such as SECONDO provide support for atomic symbolic trajectories. In SECONDO, symbolic trajectories are implemented as *moving labels*, which are temporal values whose states are symbolic labels associated with time intervals. The system supports temporal and symbolic operations, including pattern matching, rewriting, and classification of symbolic trajectories inside the database engine [14].

Support in Production Database Systems Production database systems provide more limited but highly integrated support for temporal symbolic values. MobilityDB, which extends PostgreSQL and PostGIS, provides native temporal data types defined over atomic value domains such as integers, booleans, and text [29]. In particular, the `ttext` type allows atomic symbolic trajectories to be represented as a single temporal value whose states are text labels.

For example, an activity history can be stored as follows:

```
ttext '[("home"@2023-05-01 00:00,
        "work"@2023-05-01 08:30,
        "shopping"@2023-05-01 17:30)]'
```

In this representation, each label is associated with a timestamp, and MobilityDB manages the temporal intervals between successive values internally. Such values benefit from native SQL support, temporal indexing, and generic temporal operators.

Limitations of Atomic Symbolic Trajectories Despite their practical usefulness and system-level support, atomic symbolic trajectories are limited by the fact that each symbolic state is represented as a single, indivisible label. This makes them unsuitable for representing symbolic states that contain multiple attributes, multiple aspects, or internal structure. Any additional information must either be represented externally or encoded within a single label, which obscures the structure of the symbolic state.

Existing symbolic trajectory models therefore provide an effective abstraction for representing the temporal evolution of symbolic states, but they generally assume that symbolic values are atomic. As modern applications increasingly rely on semi-structured and heterogeneous information, this assumption limits the ability of symbolic trajectories to capture rich symbolic context while preserving native temporal behavior.

These observations motivate the need to consider more expressive symbolic trajectory models while preserving the temporal abstraction that makes symbolic trajectories suitable for database support.

2.1.5.2 Structured Symbolic Trajectories

Atomic symbolic trajectories represent each symbolic state using a single, indivisible label. While this model is effective for simple symbolic values, it becomes insufficient when symbolic states need to capture richer information. In many applications, a symbolic state consists of several attributes that jointly describe context. For example, an activity may involve an activity type, a location category, and a transportation mode, while a place may be described by its name, type, and function. These cases require symbolic states with internal structure rather than simple labels.

One way to represent structured symbolic states is to decompose them into multiple attributes and store each attribute separately. In relational databases, this approach requires defining a fixed schema in which each symbolic dimension corresponds to a separate column or table. While this makes individual attributes accessible for querying, it also makes the representation rigid. All symbolic states must conform to the same schema, which makes it difficult to support heterogeneous states where different trajectory segments have different sets of attributes. Extending the schema to accommodate new symbolic properties requires schema changes and query modifications.

A common alternative is to store structured symbolic states as composite objects, for example by associating timestamps or time intervals with JSON documents. This

approach allows symbolic states to carry heterogeneous attributes without enforcing a rigid schema. PostgreSQL provides native mechanisms for querying and indexing JSON data, making individual attributes accessible for analysis. However, the JSON document and its temporal validity remain separate components of the data model. Temporal operators reason about timestamps and intervals, while JSON operators reason about the internal structure of the symbolic state. Queries must therefore combine temporal and structural conditions explicitly, rather than treating the structured symbolic state as a single temporal object.

Support for structured symbolic trajectories remains limited in existing spatio-temporal database systems. For example, MobilityDB provides native temporal types defined over atomic value domains, which support numeric, geometric, and simple symbolic values. However, structured symbolic states cannot be represented directly as native temporal values within the existing set of MobilityDB temporal types. In practice, symbolic trajectories must be represented using multiple temporal attributes or encoded as atomic values.

In the latter case, multiple attributes are combined into a single text-based symbolic value, such as `"work|office"` or `"commute|bus"`. From the database perspective, this value remains atomic. Although individual components can be extracted using string-processing techniques, they are not represented as separate symbolic attributes within the temporal model. As a result, queries involving specific symbolic components require application-level parsing or additional processing rather than direct use of native temporal operators. Temporal comparisons and indexing operate on complete values, which limits the ability to combine temporal reasoning with queries over individual symbolic dimensions.

These representations allow structured symbolic information to be stored, but they do not integrate it into the temporal model as a first-class temporal value. Query formulation becomes more complex, and symbolic reasoning remains separate from temporal reasoning. This gap motivates representations that support structured and heterogeneous symbolic states while preserving native temporal behavior.

2.1.6 Comparative Summary and Identified Research Gap

The progression from temporal data models to geometric, semantic, and symbolic trajectory models reflects an increasing ability to represent and analyze more complex forms of time-varying information. Temporal database research provides a foundation for representing and querying time-varying data through temporal semantics, interval-based representations, and generic temporal operators. Spatio-temporal databases extend these concepts to moving objects, while semantic and symbolic trajectory models introduce higher-level interpretations of movement and behavior.

At the same time, this progression reveals challenges from a database-system perspective. Temporal abstractions integrate naturally with database query processing when values are simple and indivisible, such as numbers, geometries, or atomic symbols. When symbolic states become richer and contain multiple attributes, maintaining this integration becomes more difficult.

Semantic trajectory models capture meaning through activities, places, events, and contextual information. While expressive from a conceptual perspective, they typically rely on representations that distribute information across multiple entities and relationships. In practice, this often requires multiple tables, joins, and explicit handling of temporal conditions, which increases query complexity and separates semantic reasoning from

temporal processing.

Symbolic trajectory models address part of this challenge by representing symbolic evolution as a time-varying value. This abstraction aligns naturally with temporal data models and provides a foundation for temporal querying and analysis. However, symbolic trajectories are generally defined over atomic symbolic domains. As a result, symbolic states with internal structure or heterogeneous attributes are not naturally represented within the symbolic trajectory abstraction.

When structured symbolic states must be represented, existing approaches typically rely on indirect encodings. Symbolic information may be collapsed into atomic values, decomposed into multiple attributes, or stored in semi-structured objects associated with timestamps or time intervals. While these approaches allow structured information to be stored, they do not represent the structured symbolic state itself as a native temporal value. As a result, temporal reasoning and symbolic reasoning remain only partially integrated.

Within MobilityDB, native temporal types support numeric, geometric, and atomic symbolic values. However, the existing temporal type system does not provide a native temporal type for structured symbolic values. Consequently, structured symbolic trajectories must be represented using alternative modeling approaches that fall outside the temporal type framework.

These observations motivate **RQ1**, which investigates the limitations of existing temporal data types in MobilityDB for representing structured symbolic trajectories.

This thesis addresses this limitation by extending MobilityDB with support for structured symbolic trajectories. The proposed approach builds on the temporal framework already provided by MobilityDB, including its temporal data types, operators, and indexing mechanisms. Rather than introducing a new temporal model, it extends the existing framework to support symbolic values with internal structure that evolve over time as first-class temporal objects.

The scope of this work is deliberately limited. It does not attempt to define semantic ontologies, infer symbolic meaning from raw data, or replace semantic trajectory models. Instead, it focuses on data representation, querying, and system-level integration within the database engine.

2.2 Technical Foundations

2.2.1 MobilityDB: Architecture and Capabilities

2.2.1.1 System Overview and Temporal Model

MobilityDB is an open-source database system for managing and querying temporal and spatio-temporal data [29]. It is implemented as an extension of PostgreSQL and builds on PostGIS for spatial support. This design allows MobilityDB to integrate with the PostgreSQL ecosystem while relying on its relational storage engine, query optimizer, and execution framework [21, 22]. The primary objective of MobilityDB is to provide native support for temporal and spatio-temporal data through an abstract data type (ADT) architecture that represents time-varying values as first-class database objects [29].

The core algorithms of MobilityDB are implemented in MEOS (Mobility Engine Open Source), an open-source library that provides data structures and operations for temporal and spatio-temporal objects [28]. MEOS can be used independently of PostgreSQL

in standalone applications and analytical workflows, while MobilityDB exposes this functionality through SQL types, functions, and operators integrated into PostgreSQL [19, 28].

The temporal model of MobilityDB represents time-varying data as functions from time to a value domain, which may be spatial or non-spatial. Temporal values are therefore treated as first-class objects rather than collections of timestamped records. In practice, these functions are represented using temporal instants, temporal sequences, and temporal sequence sets, together with interpolation rules appropriate for the underlying value domain [19]. This representation provides a uniform framework for representing temporal behavior across different data types.

The MobilityDB type system includes temporal types defined over numeric, boolean, textual, and spatial domains. Examples include temporal integers (`tint`), temporal floating-point values (`tfloat`), temporal booleans (`tbool`), temporal text values (`ttext`), and temporal geometries and geographies (`tgeompoint` and `tgeogpoint`) [19]. All temporal types share the same temporal model and support a common set of temporal operations.

MobilityDB has been used in a variety of research applications involving trajectory analysis, sensor data, and environmental monitoring [13, 24]. Its architecture also supports deployment within larger PostgreSQL-based systems, making it suitable for both experimental and practical data-management scenarios.

2.2.1.2 Query Processing and Indexing

MobilityDB extends SQL with native support for temporal and spatio-temporal data types. It provides functions and operators that allow temporal objects to be queried directly without decomposing them into separate value–time relations. Supported operations include accessing the value of a temporal object at a given time, restricting it to a time period, retrieving its temporal domain, and performing temporal transformations, comparisons, and aggregations [19].

For example, a trajectory can be restricted to a given time interval using a temporal restriction operation:

```
SELECT
  atPeriod(traj, tstzspan('[2024-01-01, 2024-01-31]'))
FROM trajectories;
```

Similarly, the value of a temporal object can be retrieved at a specific timestamp:

```
SELECT
  valueAtTimestamp(traj, '2024-01-15 10:00:00+00')
FROM trajectories;
```

These queries operate directly on temporal values and allow temporal behavior to be expressed within SQL using the temporal type system [19].

The semantics of temporal operations are defined consistently across MobilityDB temporal types. Operations such as restriction, comparison, and transformation are available independently of the underlying value domain. Most of the corresponding algorithms are implemented in MEOS, while MobilityDB provides SQL-level integration and interaction with PostgreSQL query planning and execution.

To support efficient query processing, MobilityDB associates temporal objects with bounding representations such as temporal spans and spatio-temporal bounding boxes.

These representations are used as filtering structures during query evaluation and can be indexed using PostgreSQL indexing mechanisms, including GiST and SP-GiST indexes [19, 22]. During query execution, bounding representations allow candidate objects to be filtered before more expensive temporal or spatio-temporal computations are performed.

This architecture is particularly relevant to the present thesis because new temporal types can be integrated into the MobilityDB framework by reusing the same temporal representations, operators, and indexing infrastructure. The proposed `tjsonb` type follows this approach by extending the set of value domains supported by MobilityDB while preserving its temporal model and query-processing architecture.

2.2.2 JSON and JSONB in PostgreSQL

2.2.2.1 Overview of JSON Support in PostgreSQL

JSON (JavaScript Object Notation) is a lightweight, text-based data exchange format that is independent of programming languages. It was originally defined to represent structured and semi-structured data in a simple and portable way [8]. A JSON document is composed of objects, defined as collections of key–value pairs, and arrays, which are ordered lists of values. These structures can be nested to represent hierarchical data.

PostgreSQL introduced native JSON support in version 9.2, providing integrated support for semi-structured data within a relational database system [22]. This support addresses the needs of applications in which data schemas may be flexible or evolve over time. The `json` type allows JSON documents to be stored directly in relational tables while preserving PostgreSQL features such as transactions, concurrency control, and security.

The `json` type stores JSON documents as validated text and preserves the original textual representation, including key order and whitespace. PostgreSQL provides functions and operators to extract values and navigate the JSON structure, making it possible to use JSON data directly in SQL queries.

The following example shows the use of the `json` type to represent a structured application state:

```
SELECT
  '{"activity":"Work",
    "location":"Office A12",
    "confidence":0.9,
    "sensors":["wifi","badge"]}'::json;
```

This document can be stored in a table column, exchanged with client applications, or partially queried using PostgreSQL JSON operators. The `json` type therefore helps bridge relational and document-oriented data models within a single database system.

2.2.2.2 JSONB: Binary Storage Model

To improve support for JSON documents, PostgreSQL version 9.4 introduced the `jsonb` (JSON Binary) type [3]. While the `json` type stores a textual representation of the document after syntax validation, `jsonb` uses an internal binary representation. The document is parsed once at insertion time and then stored in a structured binary form.

This binary representation applies content normalization. Insignificant whitespace is removed, duplicate keys are eliminated with the last occurrence kept, and object keys

are stored in a deterministic internal order. These design choices provide a canonical representation of JSON documents, which simplifies comparison and manipulation within PostgreSQL [22]. As a result, the exact original textual form of the input document is not preserved.

The `jsonb` type builds on earlier work on key–value data management in PostgreSQL, particularly the `hstore` extension [2]. While `hstore` provides efficient storage for flat key–value pairs, it does not support hierarchical structures. The `jsonb` type can therefore be viewed as a generalization of `hstore`, combining efficient storage with support for nested objects and arrays.

In practice, the `json` and `jsonb` types serve different purposes. The `json` type is useful when the exact input document must be preserved, whereas `jsonb` is generally preferred when JSON documents need to be queried, compared, and integrated with relational data.

2.2.2.3 Querying JSONB

The `jsonb` type provides PostgreSQL with mechanisms for querying semi-structured data. Unlike the `json` type, which is stored as text, `jsonb` uses an internal binary representation that allows direct access to document elements. PostgreSQL provides operators for navigation and value extraction (`->`, `->>`, `#>`, `#>>`) as well as operators such as `@>` (containment) and `?` (key existence). These operators make it possible to express predicates directly on the structure of JSONB documents [22].

These querying capabilities are particularly useful for data whose structure may vary over time. This is common in Internet of Things (IoT) settings, where events and observations are represented as JSON documents containing measurements, metadata, and contextual information, and where the set of available attributes may differ between observations. Several widely used standards and APIs in this area, such as SensorThings, rely on JSON representations to model and exchange sensor observations [17].

Starting from PostgreSQL 12, support for the SQL/JSON standard and the `jsonpath` language further extended these capabilities. `jsonpath` expressions allow declarative queries over JSON documents, including nested navigation and filtering, through operators such as `@?` and `@@` [12].

2.2.2.4 Performance Considerations

Performance considerations play an important role in the choice between the `json` and `jsonb` types, particularly when large collections of JSON documents are queried repeatedly. The `json` type relies on textual storage, which requires repeated parsing when document contents are accessed. This can increase the cost of query execution, especially when nested structures are involved.

In contrast, the `jsonb` type uses an internal binary representation. The document is parsed once at insertion time and then stored in a structured form. This design reduces the cost of repeated access to document contents and supports more direct navigation of internal elements. The trade-off is that insertion may require additional processing because parsing and normalization occur when the document is stored.

In practice, `jsonb` is commonly used when JSON documents need to be queried, filtered, or manipulated within PostgreSQL. The `json` type remains useful when preservation of the original document representation is required.

The characteristics of `jsonb` are particularly relevant to this thesis. Its ability to represent structured and heterogeneous information without requiring a fixed schema

makes it suitable for modeling symbolic states composed of multiple attributes. In addition, PostgreSQL provides native operators for accessing and querying the internal structure of `jsonb` documents. These properties make `jsonb` a suitable value domain for extending MobilityDB with support for structured symbolic trajectories.

Chapter 3

Structured Symbolic Trajectories in MobilityDB

3.1 Motivation and Representation

This section addresses **RQ2** by describing how structured symbolic trajectories can be represented in MobilityDB.

As discussed in Chapter 2, MobilityDB currently supports symbolic trajectories only for atomic value domains. While this is suitable for simple labels, it does not allow symbolic states to contain multiple attributes or structured information. As a result, structured symbolic data must be represented using indirect approaches outside the temporal type system.

MobilityDB represents temporal data as functions from time to values. Since temporal behavior is independent of the value domain, supporting structured symbolic trajectories does not require a new temporal model. Instead, it requires a value type capable of storing structured symbolic states.

The proposed solution uses PostgreSQL's native `jsonb` type as the value domain for symbolic trajectories. The `jsonb` type supports structured, nested, and heterogeneous data and allows individual attributes to be queried when needed. Using `jsonb` values makes it possible to represent activities with attributes, system states with parameters, and other forms of structured symbolic information without requiring a fixed schema.

Based on this representation, the proposed model introduces a new temporal type, `tjsonb`, whose values are `jsonb` documents. Temporal operators continue to work on the temporal dimension, while predicates over JSON attributes can be combined with temporal predicates in SQL queries.

This approach allows structured symbolic trajectories to be represented as first-class temporal objects within MobilityDB.

3.1.1 Formal Definition of the `tjsonb` Type

The `tjsonb` type represents time-dependent functions whose values are `jsonb` documents.

Formally, a `tjsonb` object represents a function

$$f : \mathcal{T} \to \mathcal{J},$$

where $\mathcal{T}$ is the temporal domain and $\mathcal{J}$ is the set of `jsonb` values.

A JSON document may be associated with a single timestamp or with one or more time intervals. As with other MobilityDB temporal types, temporal values are represented using instants, sequences, and sequence sets.

An *instant* associates a JSON document with a timestamp, for example

$$\texttt{\{"state":"A"\}}@t_1.$$

A *sequence* represents the evolution of JSON values over time, for example

$$[\ \texttt{\{"state":"A"\}}@t_1,\ \texttt{\{"state":"B"\}}@t_2\ ] \quad \text{with } t_1 < t_2.$$

A *sequence set* represents discontinuous evolution as a collection of sequences over disjoint time intervals, for example

$$\left[\begin{array}{l}[\ \texttt{\{"state":"A"\}}@t_1,\ \texttt{\{"state":"B"\}}@t_2\ ],\\ [\ \texttt{\{"state":"C"\}}@t_3,\ \texttt{\{"state":"D"\}}@t_4\ ]\end{array}\right], \quad \text{with } [t_1, t_2] \cap [t_3, t_4] = \emptyset.$$

Sequences use stepwise interpolation. A value remains constant until a new JSON document is assigned. This behavior is consistent with discrete temporal types such as `ttext`.

The temporal domain relies on the standard temporal types of PostgreSQL and MobilityDB. Individual timestamps are represented using `timestamptz`, sets of timestamps using `tstzset`, intervals using `tstzspan`, and collections of disjoint intervals using `tstzspanset`.

3.1.2 Formal Definition of the `jsonbset` Type

The `jsonbset` type represents finite sets of JSON documents. It follows the same principles as existing MobilityDB set types such as `intset`, `floatset`, and `dateset`.

Formally, a `jsonbset` value represents a finite set

$$S = \{j_1, j_2, \ldots, j_n\},$$

where each element j_i is a value of type `jsonb`.

Unlike `tjsonb`, the `jsonbset` type is not temporal and does not associate values with timestamps or intervals. Its purpose is to represent collections of JSON documents as database values.

Individual `jsonb` values and arrays of `jsonb` values can be converted into sets through constructors and casts. Set elements can also be expanded into relational form when needed.

Although `jsonbset` can be used independently, it also complements the `tjsonb` type by providing a collection-oriented representation of JSON documents.

3.2 System Integration

3.2.1 Goals and Scope

This section describes how the `jsonbset` and `tjsonb` types are integrated into MobilityDB.

The focus is on the system components required to support the new types, including storage structures, SQL interfaces, operators, aggregates, and index support. The section highlights the main architectural decisions behind the implementation, while detailed SQL definitions and function signatures are provided in the appendix.

3.2.2 System Architecture

MobilityDB is built on top of MEOS, which provides the core temporal data structures and algorithms. MEOS implements temporal objects and operations such as comparison, restriction, transformation, and aggregation. MobilityDB exposes this functionality through PostgreSQL types, functions, operators, aggregates, and operator classes.

The implementation follows the same architecture. At the MEOS level, JSON values are stored inside temporal containers together with their associated timestamps and time spans. Existing mechanisms for normalization, validation, and memory management are reused.

At the PostgreSQL level, MobilityDB defines the SQL bindings required to expose the new functionality. The `jsonbset` and `tjsonb` types are integrated through type definitions, casts, functions, operators, aggregates, and index operator classes. Query execution, planning, and index selection continue to rely on PostgreSQL's standard mechanisms.

3.2.3 Design Principles

The implementation follows three design principles.

First, JSON support is introduced by extending the value domain rather than by introducing new temporal structures. This allows structured JSON documents to be handled using the same temporal representations already available in MobilityDB.

Second, existing infrastructure is reused whenever possible. Temporal data structures, algorithms, and processing mechanisms provided by MEOS and MobilityDB are shared by the new types, reducing implementation complexity and maintaining consistency across the system.

Third, the implementation follows the conventions used by existing MobilityDB types. Type definitions, operators, aggregates, and naming schemes are aligned with those already present in the system, allowing the new types to integrate naturally with existing SQL usage patterns.

3.3 JSON Set Representation

This section describes the `jsonbset` type, which adds support for sets of JSONB values in MobilityDB. The type provides functions and operators for creating, accessing, comparing, aggregating, and manipulating sets of JSON documents.

3.3.1 The `jsonbset` Type

The `jsonbset` type represents a set of JSONB values. Values can be created from a single JSON document using `set(jsonb)` or from multiple documents using `set(jsonb[])`. Input and output functions are also provided, allowing `jsonbset` values to be read and written in both text and binary formats.

Functions such as `startValue`, `endValue`, `valueN`, and `getValues` provide access to values stored in the set. Functions such as `numValues` and `memSize` return information about the number of values and the storage size of a set. The type also supports expansion into rows using `unnest(jsonbset)` and aggregation using the `setUnion` aggregate.

3.3.2 Set Operations

The `jsonbset` type supports containment, overlap, and set operations. Containment is provided through the operators `@>` and `<@`, while overlap is provided through the operator `&&`. Union, intersection, and difference are provided through the operators `+`, `*`, and `-`, respectively.

These operations can be applied between two `jsonbset` values and, when appropriate, between a `jsonbset` value and a single JSONB value.

3.3.3 Comparison and Index Support

The `jsonbset` type supports equality, ordering, and comparison through functions such as `set_eq`, `set_lt`, `set_gt`, and `set_cmp`. These functions are available through the corresponding SQL comparison operators.

A B-tree operator class is provided for ordering and comparison operations, while a hash operator class is provided for equality operations. These operator classes allow `jsonbset` values to be used in PostgreSQL sorting, grouping, joins, and indexes.

The `jsonbset` type provides set-based operations for JSONB values and supports the temporal functionality provided by `tjsonb`.

3.4 Temporal JSON Values

The `tjsonb` type uses JSONB values as the value domain of a temporal type. A `tjsonb` value associates a JSON document with time, allowing structured states to be represented as temporal objects rather than as individual timestamped records.

Temporal JSON values can be created from JSONB values together with timestamps, timestamp sets, spans, and span sets. Time restriction functions such as `atTime` and `minusTime` allow values to be restricted to selected timestamps or time periods.

3.4.1 Temporal Structure

`tjsonb` values are represented using temporal instants, temporal sequences, and temporal sequence sets.

A temporal instant associates a JSON value with a single timestamp. Temporal sequences represent changes over a continuous time interval, while temporal sequence sets represent multiple sequences separated by gaps in time. Constructors such as `tjsonbSeq`, `tjsonbSeqSet`, and `tjsonbSeqSetGaps` are provided for creating these structures.

Accessor functions allow both the JSON values and their temporal information to be retrieved. Examples include `getValue`, `getTimestamp`, `valueAtTimestamp`, `getValues`, `getTime`, `startValue`, `endValue`, `numInstants`, and `numSequences`.

Interpolation is controlled through the functions `interp` and `setInterp`. The constructor `tjsonbSeq` uses `'step'` interpolation by default.

Functions such as `shiftTime`, `scaleTime`, `shiftScaleTime`, `merge`, and `tsample` allow temporal values to be modified or resampled.

3.4.2 Input, Output, and External Formats

The `tjsonb` type supports text, binary, hexadecimal, and MF-JSON formats.

Input functions include `tjsonbFromText`, `tjsonbFromBinary`, `tjsonbFromHexWKB`, and `tjsonbFromMFJSON`.

Output functions include `asText`, `asBinary`, `asHexWKB`, and `asMFJSON`.

3.5 Operations and Aggregates

The `tjsonb` type supports temporal operations, JSON operations, and aggregate functions. Together, these functions allow temporal JSON values to be queried, compared, combined, and summarized.

3.5.1 Temporal Predicates and Comparison Operations

The implementation provides predicates that test whether JSON keys exist in temporal JSON values. These predicates are available through the operators `?`, `?|`, and `?&`, which test for the existence of a single key, any key from a set of keys, or all keys from a set of keys.

Containment is supported through the operators `@>` and `<@`. These operators are defined between `jsonb` and `tjsonb` values as well as between two `tjsonb` values and return temporal boolean results.

The implementation also provides ever and always predicates for equality and inequality. Operators such as `?=` and `%=` test whether a condition holds at least once or throughout the whole temporal extent. Operators `?<>` and `%<>` provide the corresponding inequality tests.

Temporal equality and inequality are supported through the operators `#=` and `#<>`, which return values of type `tbool`. These operators evaluate equality and inequality over time and can be used in other temporal expressions.

3.5.2 Temporal Relationship Operators

The `tjsonb` type supports the same temporal relationship operators used for other temporal types.

Topological relationships include containment (`#@>`, `<@#`), overlap (`&&`), equality (`≅`), and adjacency (`-|-`).

Relative-position relationships include before (`«#`), overbefore (`&<#`), after (`#»`), and overafter (`#&>`).

These operators are available both between `tjsonb` values and temporal spans and between pairs of `tjsonb` values.

3.5.3 JSON-Specific Operations

The `tjsonb` type also supports JSON operations adapted to temporal values.

Field access is provided through the operators `->` and `->>`, while path-based access is provided through the operators `#>` and `#>>`.

JSON documents can be combined using the operator `||`. The functions `tjsonb_set` and `tjsonb_insert` allow JSON content inside temporal values to be modified.

JSON attributes can also be converted into temporal floating-point values using the function `tfloat`. This allows numerical data stored inside JSON documents to be used with temporal numeric functions and operators.

3.5.4 Temporal Aggregates

The implementation provides aggregate functions for combining and summarizing temporal JSON values.

The aggregate `extent` returns the temporal span covered by a set of `tjsonb` values, while `merge` combines multiple temporal values into a single `tjsonb` value.

Temporal counting aggregates are provided through `tcount` and `wcount`. The aggregate `tcount` computes temporal counts, while `wcount` computes counts using a specified temporal window.

The aggregates `appendInstant` and `appendSequence` support the construction of temporal JSON values from temporal instants and temporal sequences.

3.6 Query Processing Support

The `tjsonb` type can be used in SQL queries through operators, functions, aggregates, and indexes. As a result, temporal JSON values can be combined with standard PostgreSQL and MobilityDB queries without changes to the SQL language.

The implementation exposes these capabilities through standard SQL definitions, allowing `tjsonb` values to participate in query execution in the same way as other MobilityDB temporal types.

3.6.1 Interaction with the PostgreSQL Planner

Query planning relies on PostgreSQL's standard optimization mechanisms together with the operator classes defined for `tjsonb`. Temporal predicates, comparison operators, aggregates, and indexes are exposed through standard SQL definitions and can therefore be used during query execution and optimization.

No changes to the PostgreSQL planner are required. Integration is achieved through the definition of types, functions, operators, aggregates, and index operator classes.

3.6.2 Temporal Tiling

Temporal tiling operations available in MobilityDB can also be applied to `tjsonb` values. These operations partition temporal values into temporal intervals and can be used together with temporal aggregates and analysis functions.

No JSON-specific tiling mechanism is introduced. Temporal JSON values use the same temporal tiling operations available for other MobilityDB temporal types.

3.7 Indexing and Optimization Support

The `tjsonb` type includes index support for temporal predicates and relationship operators.

3.7.1 Temporal Indexing

A GiST operator class, `tjsonb_rtree_ops`, is defined for the `tjsonb` type. The operator class uses `tstzspan` as its index representation and supports temporal relationship operators including overlap (`&&`), containment (`#@>`, `<@#`), equality (`≅`), adjacency (`-|-`), and relative-position operators such as `«#`, `#»`, `&<#`, and `#&>`.

The implementation also provides the SP-GiST operator classes `tjsonb_quadtree_ops` and `tjsonb_kdtree_ops`. These operator classes support the same family of temporal relationship operators.

Indexing is based on temporal bounds represented by `tstzspan` values. The contents of JSON documents are not included in the index key. Consequently, indexing supports temporal relationships rather than predicates on individual JSON attributes.

Hash-based equality support is also provided through the `tjsonb_hash_ops` operator class.

3.8 Completeness and Implementation Scope

The implementation provides support for construction, access, transformation, comparison, restriction, modification, aggregation, and indexing of `jsonbset` and `tjsonb` values.

The previous sections described how these capabilities were integrated into MobilityDB through new types, operators, aggregates, and index support. Detailed SQL definitions, function signatures, and operator declarations are provided in the appendix.

The scope of this work covers data representation, temporal operations, query support, and indexing for structured symbolic trajectories. It does not address semantic interpretation of JSON content, schema inference, or automatic reasoning over JSON structure. These tasks remain the responsibility of application-level logic.

Chapter 4

Evaluation

This chapter evaluates the integration of structured symbolic trajectories into MobilityDB through the proposed `tjsonb` temporal type. The goal is to understand what this representation enables in practice, both in terms of how data can be queried and how the system behaves when processing it.

The evaluation focuses on two main aspects. The first is expressiveness, that is, how queries can be formulated when working with structured symbolic trajectories. In particular, it examines how temporal and JSON-based information can be combined within a single abstraction. The second aspect concerns system behavior, including query performance, indexing support, and storage usage, and how these compare to a relational approach based on timestamped `jsonb` data.

These aspects are studied through two research questions. **RQ3** asks whether `tjsonb` allows queries to be formulated more directly on structured temporal trajectories than a relational model. **RQ4** examines the system-level implications of this design, including query performance, indexing support, and storage usage.

To answer these questions, the evaluation considers four dimensions: query formulation, query performance, indexing and execution characteristics, and storage requirements. Together, these dimensions provide a comprehensive view of the capabilities and limitations of the proposed approach.

4.1 Evaluation Methodology

This section describes how the evaluation is carried out. It presents the dataset, the data representations, and the query workload used in the experiments. The goal is to ensure that the evaluation is both reproducible and representative of real-world IoT scenarios.

4.1.1 Experimental Environment

All experiments were executed locally on a MacBook Pro running macOS 15.4.1. The evaluation machine was equipped with an Intel processor, 8 GB of RAM, and internal SSD storage.

The experiments used PostgreSQL 17.9, MobilityDB 1.4.0, PostGIS, and TimescaleDB 2.25.2. The proposed `tjsonb` implementation was integrated into MobilityDB and MEOS on top of this environment.

The MobilityDB version used for the evaluation also included additional modifications

introduced by Prof. Esteban Zimányi[1]. In particular, the system included a standalone local import of PostgreSQL `pgtypes`, which was used as part of the implementation environment.

All benchmark queries were executed on the same machine and under the same database configuration. Before measuring execution time, each query was executed once as a warm-up run. Reported execution times correspond to the median of five measured executions of the same query.

4.1.2 IoT Telemetry Use Case

The evaluation focuses on an IoT telemetry scenario in which sensors continuously produce structured observations over time. Each sensor measurement represents the state of a device at a given timestamp and includes several attributes such as temperature, humidity, light intensity, and battery voltage.

This type of data naturally forms a structured symbolic trajectory, where the state of an object evolves over time. In practice, many IoT applications require queries that combine temporal conditions with conditions on structured values.

Typical examples include:

- finding sensors active during a given time interval,
- detecting sensors whose temperature exceeded a threshold,
- computing averages over time,
- retrieving the first or latest observed state,
- analyzing how measurements evolve during specific periods.

These operations are common in monitoring systems, smart buildings, industrial sensor platforms, and environmental observation systems.

This use case was chosen because it directly reflects the main goal of the proposed `tjsonb` model: representing and querying structured states that evolve over time.

4.1.3 Dataset

The evaluation is based on a real-world IoT dataset collected from 54 sensors at the Intel Berkeley Research Laboratory between February and April 2004.[2] The data shows typical characteristics of sensor systems, such as irregular sampling, missing values, and differences in how attributes are distributed.

Each observation includes a timestamp, a sensor identifier, and several measured attributes, such as temperature, humidity, light intensity, and battery voltage. These attributes together form a structured state, stored as a JSON document.

The original dataset contains approximately 2.2 million observations. To study how the system behaves at larger scales, additional datasets are created by replicating the data. The medium dataset contains around 11 million observations, and the large dataset around 22 million observations.

[1] `https://github.com/estebanzimanyi/MobilityDB/tree/jsontypes`

[2] `https://www.kaggle.com/datasets/divyansh22/intel-berkeley-research-lab-sensor-data`

The replication process preserves the original measurement values while creating additional sensor identifiers and applying small timestamp shifts. This approach increases dataset size while avoiding duplicate sensor identifiers and timestamps.

4.1.4 Data Representation

Two different data representations are used in this evaluation in order to compare their impact on query execution and system behavior.

In the relational model, each observation is stored as a separate row. Each row contains a timestamp, a sensor identifier, and a `jsonb` value containing the measured attributes. In this representation, temporal behavior is represented through individual timestamped observations and must be reconstructed during query execution using filtering, grouping, ordering, or aggregation operations.

In the `tjsonb` model, observations are grouped by sensor and ordered in time to form a temporal trajectory for each sensor. Each trajectory is stored as a temporal object that represents the evolution of a structured state over time. This follows the MobilityDB model, where temporal information is represented directly as part of the data.

Unlike the relational representation, where each observation remains an independent row, the `tjsonb` representation materializes the temporal dimension as a single temporal object for each sensor by combining ordered observations into a temporal sequence.

Both representations encode the same sensor observations and support equivalent queries for the evaluation workload. The difference lies in how the data is organized and how temporal behavior is represented. The relational model operates on individual observations, whereas the `tjsonb` model operates directly on temporal trajectories.

4.1.5 Indexes

Indexes were created after data ingestion in order to separate data loading performance from index maintenance overhead during insertion.

In the relational model, B-tree indexes were created on timestamps to support temporal filtering. Additional functional indexes were created on frequently queried JSON attributes, such as temperature. A composite B-tree index on `(sensor_id, ts)` was also created to support queries that retrieve ordered observations for individual sensors.

In the `tjsonb` model, a GiST index was created on the trajectory column. This index supports temporal predicates and relationship operators defined for temporal values.

These indexes were used throughout the query evaluation phase of the benchmark.

4.1.6 Benchmark Design

The benchmark was designed to evaluate both the expressiveness and the system behavior of the proposed `tjsonb` model.

Rather than focusing only on synthetic microbenchmarks, the evaluation uses queries that correspond to common analytical tasks in IoT telemetry systems. The goal is to study how the two data representations behave under realistic operations involving time-series sensor data.

The benchmark includes ten query families covering different categories of operations:

- temporal filtering,

- attribute filtering,
- combined temporal and attribute predicates,
- aggregation,
- endpoint access,
- duration computation,
- window-based analysis.

These query families were selected because they represent the main types of operations commonly performed on symbolic trajectories and time-series sensor data. They also make it possible to evaluate different aspects of the systems, including temporal indexing, aggregation behavior, query formulation complexity, and trajectory processing.

The benchmark was designed to compare two fundamentally different approaches. The relational model processes many independent observations stored as rows, while the `tjsonb` model operates directly on complete trajectories. The selected queries therefore highlight how these structural differences affect both query formulation and execution behavior.

Table 4.1 summarizes the benchmark workload used in the evaluation.

Table 4.1: Benchmark query families

Query	**Description**
F1	Temporal window filtering
F2	Attribute threshold filtering
F3	Combined temporal and attribute filtering
F4	Counting observations per sensor
F5	Duration computation
F6	Global time range computation
F7	Latest value retrieval
F8	First value retrieval
F9	Average aggregation per sensor
F10	Window-based aggregation

4.2 Comparative Evaluation

This section compares the proposed `tjsonb` model with a relational approach based on timestamped `jsonb` observations stored in TimescaleDB. The objective is to evaluate how both representations behave for semantic temporal IoT workloads in terms of query formulation, ingestion, storage, and query execution performance.

TimescaleDB[3] was selected as the relational baseline because it is a widely used PostgreSQL extension for time-series management and represents a practical approach for storing timestamped JSON observations using relational structures.

The comparison evaluates two different representation models. The relational approach stores individual timestamped observations as rows, whereas `tjsonb` stores semantically

[3] `https://github.com/timescale/timescaledb`

grouped temporal trajectories as compact temporal objects. Although both approaches preserve the same information and support equivalent analyses, they differ in how temporal semantics are modeled and processed internally.

The benchmark workload includes common temporal IoT operations such as temporal filtering, semantic threshold predicates, aggregation, and temporal window queries. To ensure reproducibility, all benchmark scripts, SQL workloads, indexing strategies, and plotting utilities are publicly available in the accompanying artifact repository[4].

4.2.1 Comparison of Data Models

The main difference between the two approaches is how time is represented.

In the relational model, each observation is stored as a separate row with a timestamp and a `jsonb` value. Time is therefore spread across many rows, and any temporal behavior must be reconstructed during query execution. This typically requires filtering, grouping, or ordering operations.

In the `tjsonb` model, all observations of a sensor are grouped into a single trajectory. Time is stored directly inside the data as part of the object. This means that the evolution of values is already structured and does not need to be rebuilt during queries.

As a result, the relational model works at the level of individual observations, while the `tjsonb` model works at the level of complete trajectories. This leads to different query patterns: one based on processing many rows, and the other based on operating on fewer but more complex objects.

4.2.2 Query Formulation Comparison

This section illustrates how common analytical tasks are formulated in the two representations. The objective is not to compare performance, but to compare the resulting query formulations.

For example, selecting sensors active during a given time interval requires filtering observations in the relational representation:

```
SELECT DISTINCT sensor_id
FROM readings_timescale_small
WHERE ts >= '2004-03-01'
AND ts <= '2004-03-02';
```

In the `tjsonb` representation, the same condition is expressed as a temporal predicate over trajectories:

```
SELECT sensor_id
FROM readings_mobility_small
WHERE traj && tstzspan('[2004-03-01, 2004-03-02]');
```

The difference is also visible when temporal and attribute-based conditions are combined. In the relational representation, both conditions are evaluated on individual observations:

```
SELECT DISTINCT sensor_id
FROM readings_timescale_small
WHERE ts >= '2004-03-01'
```

[4] https://github.com/leilabourouf/tjsonb-benchmark

```
AND ts <= '2004-03-02'
AND (data->>'temperature')::float > 30;
```

In the `tjsonb` representation, the same query is formulated using a temporal predicate together with a condition on a temporal attribute extracted from the trajectory:

```
SELECT sensor_id
FROM readings_mobility_small
WHERE traj && tstzspan('[2004-03-01, 2004-03-02]')
AND tfloat(traj, 'temperature') ?> 30;
```

The `tjsonb` representation also allows JSON attributes to participate directly in temporal processing. For example, the temperature attribute can be extracted as a temporal float value and queried using temporal operators:

```
SELECT sensor_id
FROM readings_mobility_small
WHERE tfloat(traj, 'temperature') ?> 30;
```

Aggregation illustrates a similar distinction. In the relational representation, aggregation is performed over collections of observations:

```
SELECT sensor_id,
AVG((data->>'temperature')::float)
FROM readings_timescale_small
GROUP BY sensor_id;
```

In the `tjsonb` representation, aggregation is applied to a temporal attribute derived from the trajectory:

```
SELECT sensor_id,
twAvg(tfloat(traj, 'temperature'))
FROM readings_mobility_small;
```

Access to specific states also differs between the two representations. In the relational representation, retrieving the latest value requires aggregation or ordering:

```
SELECT sensor_id,
last((data->>'temperature')::float, ts)
FROM readings_timescale_small
GROUP BY sensor_id;
```

In the `tjsonb` representation, the latest state is obtained directly from the trajectory:

```
SELECT sensor_id,
(endValue(traj)->>'temperature')::float
FROM readings_mobility_small;
```

The trajectory representation also supports operations that preserve temporal objects. For example, a temporal restriction returns a trajectory restricted to a given time interval:

```
SELECT sensor_id,
atTime(traj, tstzspan('[2004-03-01, 2004-03-02]'))
FROM readings_mobility_small;
```

In the relational representation, the corresponding operation produces a set of observations rather than a temporal object.

These examples illustrate the main contribution of the `tjsonb` representation: structured symbolic states and their temporal evolution can be queried directly as temporal trajectories. Temporal predicates, temporal aggregations, and attribute-based temporal conditions are expressed on the trajectory itself rather than on collections of individual observations.

4.2.3 Ingestion Cost

This section compares the time required to load data in the relational representation and in the `tjsonb` representation. The objective is to evaluate how the two representations behave during data ingestion.

In the relational representation, each observation is inserted as a separate row containing a timestamp, a sensor identifier, and a `jsonb` value. In the `tjsonb` representation, observations are first grouped by sensor, ordered by time, and combined into temporal trajectories before insertion.

Indexes were not created before ingestion in either system. This allows the benchmark to measure ingestion time without the additional cost of index maintenance during data loading.

Table 4.2 shows the ingestion time for both representations across different dataset sizes.

Table 4.2: Ingestion time comparison (in seconds)

System	Small	Medium	Large
TimescaleDB	143.60	812.12	1618.27
MobilityDB	56.44	278.81	651.76

Figure 4.1 shows how ingestion time changes as the dataset size increases.

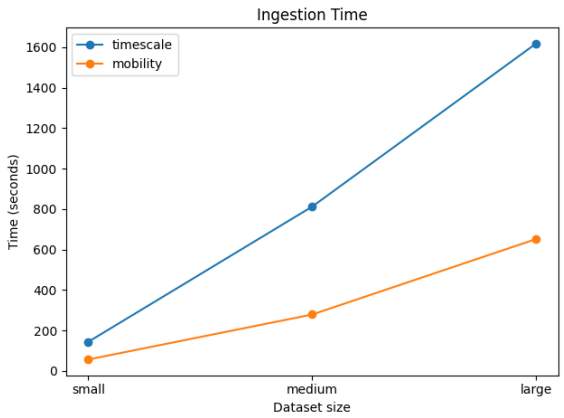

Figure 4.1: Ingestion time across dataset sizes.

Both systems exhibit increasing ingestion times as the dataset size grows. The relational representation inserts one database row for each observation, whereas the `tjsonb` representation inserts one temporal trajectory per sensor.

For the datasets used in this evaluation, the `tjsonb` representation required less ingestion time than the relational representation at all tested scales. For example, the large relational dataset contains more than 22 million observations, whereas the corresponding `tjsonb` dataset contains 550 temporal trajectories.

These results show that the two representations exhibit different ingestion characteristics. The relational representation stores observations individually, whereas the `tjsonb` representation stores grouped temporal trajectories.

4.2.4 Query Performance Comparison

This section compares query execution performance between the `tjsonb` representation (MobilityDB) and a relational representation based on timestamped `jsonb` values (TimescaleDB). The evaluation considers analytical workloads over temporally ordered IoT sensor data represented as structured JSON observations.

The benchmark includes temporal filtering, attribute threshold predicates, combined temporal and attribute filtering, grouped aggregation, endpoint retrieval, and window-based aggregation. Each query was executed on small, medium, and large datasets. Query execution times correspond to the median runtime over five executions after one warmup run.

These queries were selected to cover different aspects of analytical workloads over temporal sensor data. Some queries focus primarily on temporal filtering, whereas others emphasize aggregation, endpoint retrieval, or combined temporal and attribute-based conditions. Together, they provide a representative view of how both systems behave across a range of common IoT analysis tasks.

Temporal window queries (F1) show comparable performance between both systems. TimescaleDB is slightly faster for the small dataset, while the difference becomes smaller for the medium and large datasets.

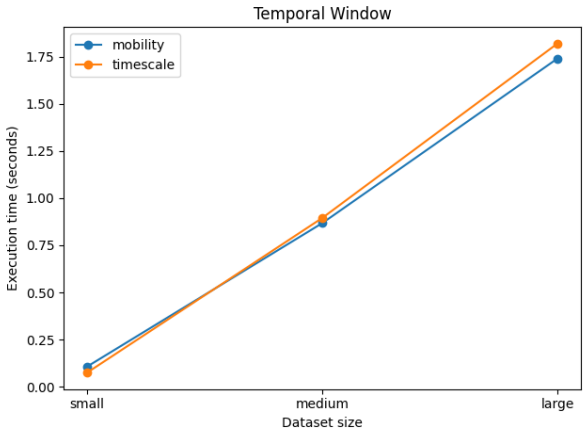

Figure 4.2: Temporal window queries (F1).

Threshold filtering (F2) consistently favors TimescaleDB. The relational representation achieves lower execution times across all dataset sizes. The benchmark configuration includes an expression index on the temperature attribute in the relational representation. This indexing support is likely a contributing factor to the lower execution times observed for this query family.

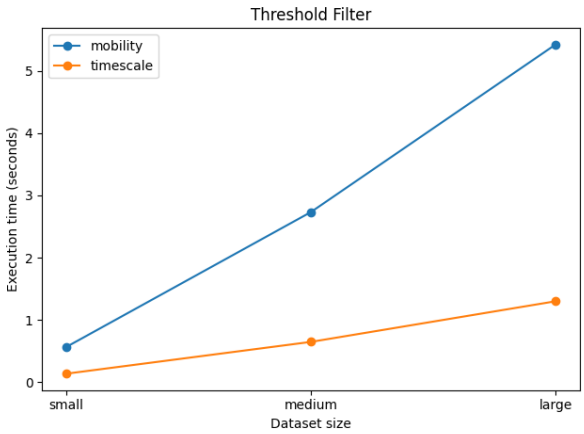

Figure 4.3: Threshold filtering (F2).

Combined time and attribute filtering (F3) produces relatively similar performance in both systems. TimescaleDB remains slightly faster, but the difference remains small across all dataset sizes.

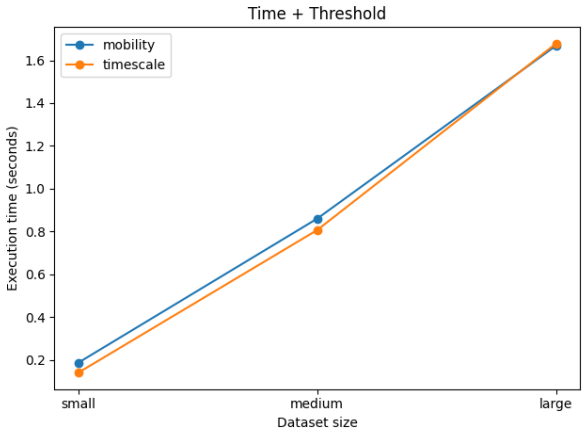

Figure 4.4: Combined filtering (F3).

Counting observations per sensor (F4) shows comparable performance between both approaches. TimescaleDB performs slightly better for the medium dataset, whereas MobilityDB performs slightly better for the large dataset.

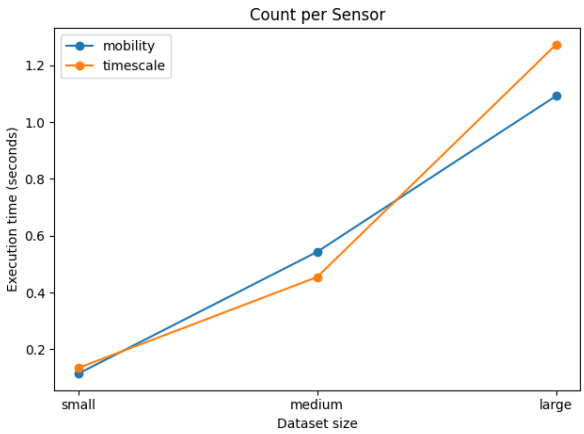

Figure 4.5: Counting per sensor (F4).

Duration computation (F5) shows similar performance in both systems, with MobilityDB achieving slightly lower execution times for the medium and large datasets.

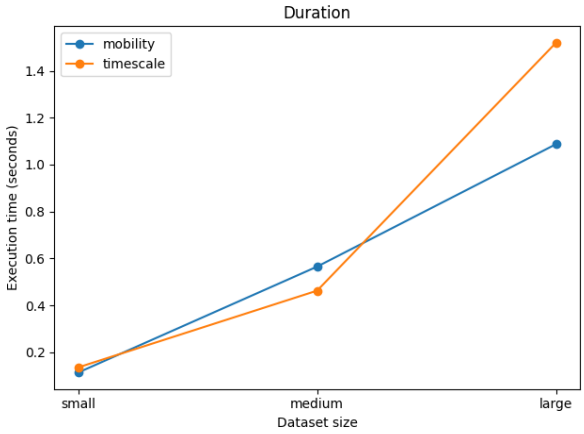

Figure 4.6: Duration queries (F5).

Global time range queries (F6) are substantially faster in TimescaleDB across all dataset sizes. The execution times remain nearly constant as the dataset size increases.

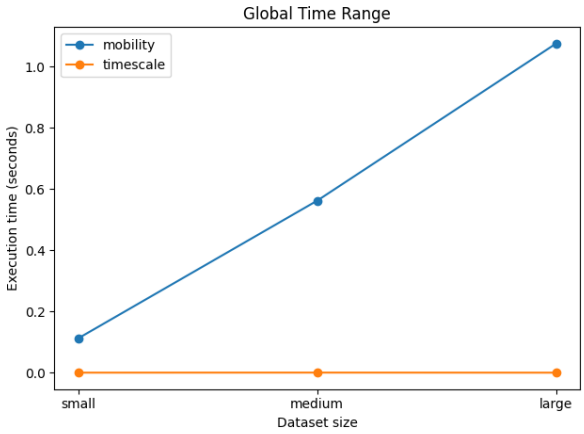

Figure 4.7: Global time range (F6).

Endpoint queries (F7–F8) highlight different performance characteristics of the two representations. MobilityDB achieves lower execution times for latest value retrieval (F7), whereas TimescaleDB achieves substantially lower execution times for first value retrieval (F8). Inspection of the query plans shows that the relational implementation of F8 uses the composite `(sensor_id, ts)` index through an index-based access path, allowing the first observation of each sensor to be retrieved efficiently. In contrast, F7 benefits from direct access to trajectory endpoints through `endValue`.

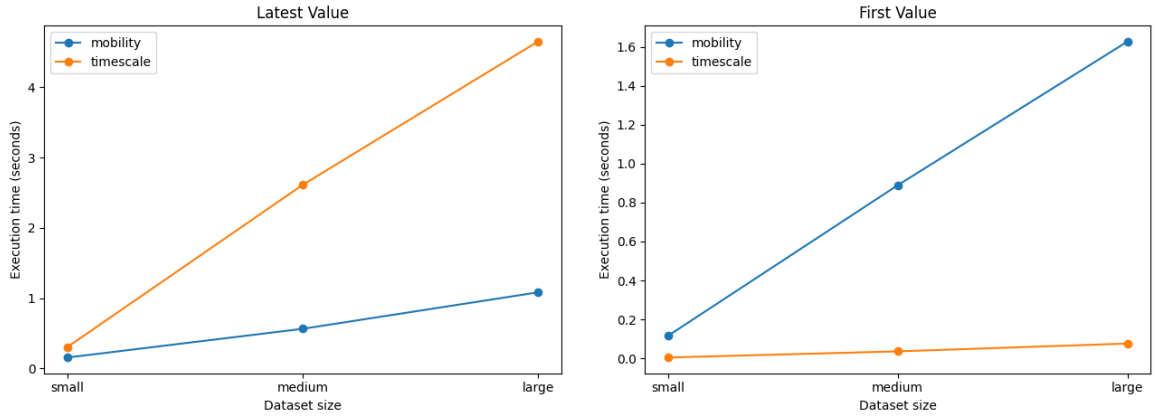

Figure 4.8: Endpoint queries (F7–F8).

Average aggregation per sensor (F9) favors TimescaleDB across all dataset sizes. Nevertheless, both systems exhibit similar scaling behavior as the dataset size increases.

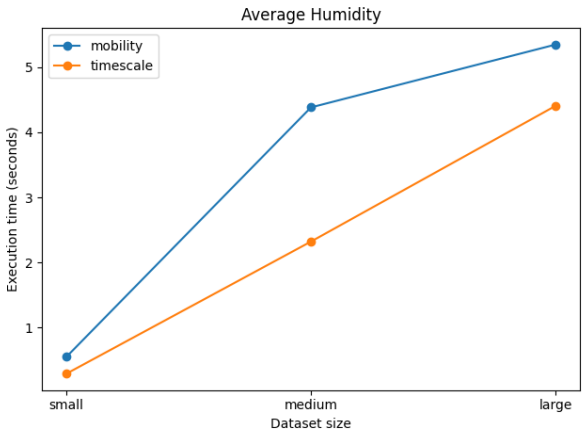

Figure 4.9: Average humidity (F9).

Window-based aggregation (F10) also favors TimescaleDB, with the difference becoming more pronounced for larger datasets.

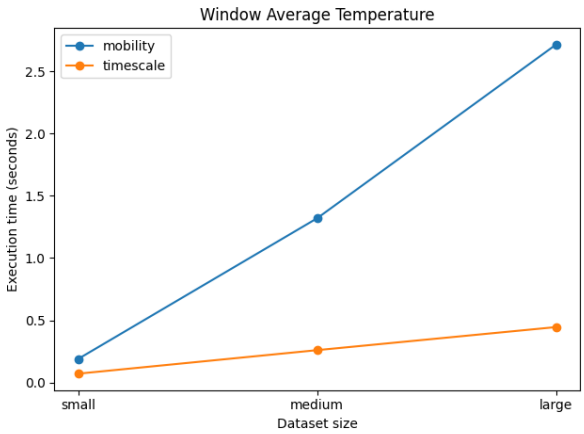

Figure 4.10: Windowed aggregation (F10).

Overall, the results show that the two systems exhibit different performance characteristics across the evaluated query families. TimescaleDB generally achieves lower execution times for threshold filtering, global time-range queries, and aggregation workloads. MobilityDB achieves comparable performance for several temporal query families and lower execution times for some trajectory-oriented operations, such as latest-value retrieval.

The results indicate that the proposed `tjsonb` representation can support analytical queries over structured temporal trajectories while maintaining competitive performance for several query families. At the same time, the evaluation shows that performance depends on the type of query being executed and on the available indexing mechanisms.

4.2.5 Indexing and Execution Behavior

The two systems use different indexing strategies.

In the relational representation, B-tree indexes were created on timestamps, expression indexes were created on extracted temperature values, and a composite index was created on `(sensor_id, ts)`. These indexes support temporal filtering, attribute-based filtering, and ordered retrieval of observations.

In the `tjsonb` representation, a GiST index was created on the trajectory column. This index supports temporal relationship predicates by indexing the temporal bounds of trajectories.

The benchmark results show that the two systems exhibit different performance characteristics across query families. TimescaleDB achieves lower execution times for several filtering and aggregation workloads, whereas MobilityDB achieves comparable performance for several temporal query families and lower execution times for some trajectory-oriented operations, such as latest-value retrieval.

These results should be interpreted together with the different data representations and indexing configurations used in the evaluation. The relational representation provides direct indexing support for selected JSON attributes and timestamps, whereas the `tjsonb` representation indexes temporal trajectories as complete temporal objects.

4.2.6 Example Query Plans

To better understand the observed performance, representative query plans were inspected using `EXPLAIN ANALYZE`.

For temporal window queries on the `tjsonb` representation, PostgreSQL selected a sequential scan under the default planner configuration. However, when sequential scans were disabled, PostgreSQL used the GiST index defined on the trajectory column:

```
Index Scan using idx_traj_large
  Index Cond:
    (traj && tstzspan(...))
```

This confirms that the GiST operator class supports temporal overlap predicates and can be used for temporal filtering on `tjsonb` values.

For first-value retrieval in the relational representation, PostgreSQL used an index-based access path through the composite `(sensor_id, ts)` index:

```
Custom Scan (SkipScan)
  -> Index Scan using
     idx_sensor_ts_large
```

This plan allows the first observation of each sensor to be retrieved efficiently from the index and helps explain the low execution times observed for query family F8.

For attribute threshold queries on the `tjsonb` representation, PostgreSQL selected a sequential scan. This behavior is expected because the GiST index stores temporal bounds and does not index individual JSON attributes. Predicates involving extracted JSON values must therefore be evaluated after the relevant trajectories have been retrieved.

These observations are consistent with the benchmark results and illustrate how the available indexing structures influence query execution in the two representations.

4.2.7 Storage Analysis

This section compares the storage requirements of the `tjsonb` representation and the relational `jsonb` representation. The objective is to evaluate how the two data models differ in terms of table size, index size, and TOAST storage usage.

Storage was measured using PostgreSQL system functions. For the relational representation, the reported size includes all hypertable chunks and indexes, reflecting the complete physical storage footprint on disk.

Table 4.3: Storage footprint comparison (in MB)

System	Dataset	Total	Table	Index	TOAST
MobilityDB	Small	45	≈0	≈0	45
MobilityDB	Medium	226	≈0	≈0	226
MobilityDB	Large	452	≈0	≈0	451
TimescaleDB	Small	530	350	179	≈0
TimescaleDB	Medium	2682	1820	861	≈1
TimescaleDB	Large	5354	3641	1713	≈1

The results show a substantial difference between the two representations.

For all dataset sizes, the `tjsonb` representation requires significantly less storage than the relational representation. Most of the storage used by MobilityDB appears in TOAST relations, whereas the main table and index sizes remain very small.

In contrast, the relational representation occupies substantially more storage in both table and index structures. The difference becomes more pronounced as the dataset size increases. For the large dataset, the relational representation requires more than 5 GB of storage, including approximately 1.7 GB of indexes.

TOAST usage remains minimal in the relational representation, while it accounts for nearly all storage used by the `tjsonb` representation.

Overall, the results show that the two representations exhibit markedly different storage characteristics. The `tjsonb` representation stores the evaluated datasets using substantially less disk space, whereas the relational representation occupies more storage in both table and index structures.

4.2.8 Trade-offs and Limitations

The evaluation highlights several differences between the `tjsonb` representation and the relational representation.

The query performance results show that neither representation is uniformly faster across all query families. TimescaleDB achieves lower execution times for several filtering and aggregation workloads, whereas MobilityDB achieves comparable performance for a number of temporal queries and lower execution times for some trajectory-oriented operations, such as latest-value retrieval.

The evaluation also reveals differences in query formulation. In the relational representation, queries are formulated over collections of timestamped observations. In the `tjsonb` representation, queries are formulated directly over temporal trajectories and temporal attributes derived from those trajectories. As shown in Section 4.2.2, temporal predicates, attribute-based conditions, and temporal aggregations can be expressed using the trajectory abstraction provided by `tjsonb`.

The ingestion and storage results demonstrate additional differences between the two representations. The `tjsonb` representation groups observations into trajectories before insertion, whereas the relational representation stores observations individually. The storage evaluation shows that the `tjsonb` representation requires substantially less disk space for the evaluated datasets, while the relational representation occupies more storage in both table and index structures.

The benchmark also highlights a limitation of the current implementation. While temporal trajectories can be indexed through the GiST index on the trajectory column, the evaluation does not include dedicated indexing support for individual JSON attributes contained within `tjsonb` values.

Overall, the evaluation shows that `tjsonb` provides a database representation for structured symbolic trajectories that integrates temporal semantics and structured JSON values within a single temporal object. The results demonstrate that this representation supports analytical querying over structured temporal trajectories while maintaining competitive performance across the evaluated workloads. Although execution times vary across query families, the benchmark shows that `tjsonb` achieves performance that is often comparable to the relational representation and performs well for several trajectory-oriented operations.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis studied how structured symbolic trajectories can be supported as native temporal objects inside MobilityDB. The work was motivated by the fact that current symbolic trajectory support in MobilityDB is limited to atomic values such as text labels, which makes it difficult to represent symbolic states that contain multiple attributes or heterogeneous information.

To address this limitation, the thesis proposed a new temporal type, `tjsonb`, based on PostgreSQL's `jsonb` type. The proposed approach allows structured JSON documents to evolve over time while preserving the temporal semantics and query model already provided by MobilityDB.

The implementation introduced two new types, `jsonbset` and `tjsonb`, and integrated them into MobilityDB and MEOS using the existing temporal architecture. The system reuses the same temporal structures, operators, aggregates, and indexing mechanisms already used for other temporal types. As a result, structured symbolic trajectories can be manipulated as first-class temporal objects using standard SQL queries.

The implementation and evaluation answer the research questions introduced in Chapter 1.

RQ1 asked about the limitations of existing temporal data types in MobilityDB for representing structured symbolic trajectories. The state-of-the-art analysis showed that current temporal symbolic types only support atomic values. When symbolic states contain multiple attributes, users must rely on workarounds such as concatenated labels, multiple temporal attributes, or relational decompositions. These approaches reduce integration with temporal operators and indexing mechanisms and make structured symbolic information more difficult to manage within the temporal framework.

RQ2 asked how structured symbolic trajectories could be represented while preserving MobilityDB's temporal semantics and query model. The thesis answered this question through the design and implementation of the `tjsonb` type. The proposed model keeps the existing temporal framework unchanged and extends only the value domain by using `jsonb` documents as temporal values. This allows structured symbolic states to be queried directly while remaining compatible with MobilityDB temporal operators and indexes.

RQ3 asked whether the `tjsonb` model improves query expressiveness and clarity compared to a relational representation based on timestamped `jsonb` data. The evaluation showed that queries in the `tjsonb` model can be formulated directly on temporal trajectories. Temporal and JSON conditions can be expressed on a single temporal object,

without requiring reconstruction of temporal behavior from individual rows. This allows structured symbolic trajectories to be queried through the same temporal abstraction used for other MobilityDB temporal types.

RQ4 asked about the system-level implications of the proposed model. The experimental evaluation showed that performance depends on the type of query. The relational model achieved lower execution times for selective filtering and several aggregation workloads. The evaluation was conducted using a configuration that included indexes on timestamps and selected JSON attributes in the relational representation. However, the `tjsonb` model achieved competitive performance for several temporal query families and lower execution times for some trajectory-oriented operations, such as retrieving the latest value of a trajectory. The evaluation also showed that the `tjsonb` representation requires substantially less storage space for the evaluated datasets.

Overall, the thesis shows that structured symbolic trajectories can be integrated into MobilityDB without changing its temporal foundations. The proposed approach extends symbolic trajectory support from atomic values to structured JSON documents while preserving MobilityDB's existing temporal semantics and query model. The resulting `tjsonb` type enables structured symbolic trajectories to be represented and queried as first-class temporal objects within the database system.

5.2 Future Work

Several directions can extend the work presented in this thesis.

A first direction concerns indexing support for JSON attributes inside temporal objects. In the current implementation, GiST and SP-GiST indexes operate only on the temporal dimension of trajectories. JSON predicates are evaluated after the temporal filtering step. Future work could investigate hybrid indexing approaches that combine temporal bounds with indexing of selected JSON fields in order to improve the performance of attribute-based queries.

Another direction concerns temporal pattern matching over structured symbolic trajectories. Existing symbolic trajectory research includes pattern languages and sequence matching techniques, but these mechanisms are not currently available for `tjsonb`. Extending MobilityDB with temporal pattern operators for structured JSON trajectories would support more advanced behavior analysis and event detection.

A third direction concerns extending the set of JSON-specific operations available for temporal values. The current implementation focuses on representation, temporal integration, and query support. Additional operators for JSON processing and richer interactions between JSON attributes and temporal types could further expand the functionality of the `tjsonb` model.

Further evaluation on larger datasets and distributed environments would also be useful. The experiments in this thesis used replicated IoT datasets on a single database instance. Evaluating `tjsonb` in distributed deployments of MobilityDB or in streaming environments would provide additional insight into scalability and real-world performance.

Finally, the ideas developed in this thesis are not limited to IoT telemetry. Structured symbolic trajectories could also be applied to domains such as human mobility analysis, transportation systems, industrial monitoring, network behavior analysis, or context-aware applications. Exploring these application domains would help better understand the practical benefits and limitations of the proposed model.

Bibliography

[1] Luis Otavio Alvares, Vania Bogorny, Bart Kuijpers, Jose Antonio Fernandes de Macedo, Bart Moelans, and Alejandro Vaisman. A model for enriching trajectories with semantic geographical information. In *Proceedings of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems (GIS '07)*, pages 22:1–22:8. ACM, 2007. doi:10.1145/1341012.1341041.

[2] Oleg Bartunov and Teodor Sigaev. Nested hstore with array support. In *PGCon*, 2013.

[3] Oleg Bartunov and Teodor Sigaev. JSONB: Binary JSON storage in PostgreSQL. In *PGCon*, 2014.

[4] Vania Bogorny, Chiara Renso, Artur Ribeiro de Aquino, Fernando de Lucca Siqueira, and Luis Otavio Alvares. CONSTAnT – a conceptual data model for semantic trajectories of moving objects. *Transactions in GIS*, 18(1):66–88, 2014. doi:10.1111/tgis.12011.

[5] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. Database technology for processing temporal data. In *Proceedings of the 25th International Symposium on Temporal Representation and Reasoning (TIME 2018)*, volume 120 of *LIPIcs*, pages 2:1–2:7. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.TIME.2018.2.

[6] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. Temporal data management – an overview. In *Encyclopedia of Big Data Technologies*, pages 51–83. Springer, 2018. doi:10.1007/978-3-319-96655-7_3.

[7] Lei Chen, M. Tamer Özsu, and Vincent Oria. Symbolic representation and retrieval of moving object trajectories. In *Proceedings of the 6th ACM SIGMM International Workshop on Multimedia Information Retrieval (MIR '04)*, pages 227–234. ACM, 2004. doi:10.1145/1026711.1026749.

[8] Douglas Crockford. The application/json media type for javascript object notation (JSON). Technical Report RFC 4627, IETF, 2006.

[9] Maria Luisa Damiani, Hamza Issa, Ralf Hartmut Güting, and Fabio Valdes. Hybrid queries over symbolic and spatial trajectories: A usage scenario. In *Proceedings of the IEEE 15th International Conference on Mobile Data Management (MDM)*, pages 341–344. IEEE, 2014. doi:10.1109/MDM.2014.49.

[10] Somayeh Dodge, Patrick Laube, and Robert Weibel. Movement similarity assessment using symbolic representation of trajectories. *International Journal of Geographical Information Science*, 26(9):1563–1588, 2012. doi:10.1080/13658816.2011.630003.

[11] Somayeh Dodge, Robert Weibel, and Ehsan Forootan. Revealing the physics of movement: Comparing the similarity of movement characteristics of different types of moving objects. *Computers, Environment and Urban Systems*, 33(6):419–434, 2009. doi:10.1016/j.compenvurbsys.2009.07.008.

[12] Nikita Glukhov and Oleg Bartunov. SQL/JSON and JSONPath in PostgreSQL. In *PGConf.EU*, 2019.

[13] Leticia I. Gómez, Alejandro A. Vaisman, and Esteban Zimányi. Querying mobile pollution data using MobilityDB. In *Proceedings of the 25th IEEE International Conference on Mobile Data Management (MDM 2024)*, pages 227–234. IEEE, 2024. doi:10.1109/MDM61037.2024.00047.

[14] Ralf Hartmut Güting, Thomas Behr, and Christian Düntgen. SECONDO: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Engineering Bulletin*, 33(2):56–63, 2010.

[15] Ralf Hartmut Güting and Markus Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005. doi:10.1016/B978-0-12-088799-6.X5000-2.

[16] Ralf Hartmut Güting, Fabio Valdés, and Maria Luisa Damiani. Symbolic trajectories. *ACM Transactions on Spatial Algorithms and Systems*, 1(2):7:1–7:51, 2015. doi:10.1145/2786756.

[17] Atanas Kotsev, Kai Schleidt, Steve Liang, Hylke Van Der Schaaf, Tayebeh Khalafbeigi, Sébastien Grellet, Michael Lutz, Simon Jirka, and Michel Beaufils. Extending INSPIRE to the internet of things through SensorThings API. *Geosciences*, 8(6):221, 2018. doi:10.3390/geosciences8060221.

[18] Ronaldo dos Santos Mello, Vania Bogorny, Luis Otavio Alvares, Luiz Henrique Zambom Santana, Carlos Andres Ferrero, Angelo Augusto Frozza, Geomar Andre Schreiner, and Chiara Renso. MASTER: A multiple aspect view on trajectories. *Transactions in GIS*, 23(4):805–822, 2019. doi:10.1111/tgis.12526.

[19] MobilityDB Development Team. *MobilityDB User Manual*, 2025. https://docs.mobilitydb.com.

[20] Christine Parent, Stefano Spaccapietra, Chiara Renso, Gennady Andrienko, Natalia Andrienko, Vania Bogorny, Maria Luisa Damiani, Aris Gkoulalas-Divanis, Jose Antonio de Macedo, Nikos Pelekis, Yannis Theodoridis, and Zhixian Yan. Semantic trajectories modeling and analysis. *ACM Computing Surveys*, 45(4):42:1–42:32, 2013. doi:10.1145/2501654.2501656.

[21] PostGIS Project Steering Committee. *PostGIS Documentation*, 2024. https://postgis.net/documentation/.

[22] PostgreSQL Global Development Group. *PostgreSQL Documentation*, 2024. https://www.postgresql.org/docs/.

[23] Gilbert Pradel and Philippe Hoppenot. Symbolic trajectory description in mobile robotics. *Journal of Intelligent and Robotic Systems*, 45:157–180, 2006. doi:10.1007/s10846-005-9027-z.

[24] Giulia Rovinelli, Davide Rocchesso, Marta Simeoni, Esteban Zimányi, and Alessandra Raffaetà. Spatiotemporal characterisation of underwater noise through semantic trajectories. *GeoInformatica*, 29:845–876, 2025. doi:10.1007/s10707-025-00547-x.

[25] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.

[26] Stefano Spaccapietra, Christine Parent, Maria Luisa Damiani, Jose Antonio de Macedo, Fabio Porto, and Christelle Vangenot. A conceptual view on trajectories. *Data & Knowledge Engineering*, 65(1):126–146, 2008. doi:10.1016/j.datak.2007.10.008.

[27] Fabio Valdés and Ralf Hartmut Güting. Index-supported pattern matching on symbolic trajectories. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 53–62. ACM, 2014. doi:10.1145/2666310.2666402.

[28] Esteban Zimanyi, Mariana Duarte, and Víctor Diví. MEOS: An open source library for mobility data management. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2024. doi:10.48786/edbt.2024.76.

[29] Esteban Zimányi, Mahmoud Sakr, Arthur Lesuisse, and Mohamed Bakli. MobilityDB: A mainstream moving object database system. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases (SSTD '19)*, pages 206–209. ACM, 2019. doi:10.1145/3340964.3340991.

Appendix A

Complete API of Structured Symbolic Trajectories

A.1 `tjsonb` Type Reference

A.1.1 Constructors

The `tjsonb` constructors allow the creation of temporal objects from static JSONB values and temporal domains. They are used to create temporal instants, generate sequences or sequence sets from temporal intervals, and assemble existing instants or sequences into coherent temporal objects, they also allows to control temporal semantics (discrete or step) and the handling of discontinuities.

`tjsonb(jsonb, timestamptz)` → `tjsonb`. Creates a temporal instant by associating a JSONB value with a single timestamp.

```
SELECT tjsonb('{"state":"A"}'::jsonb,
              '2025-01-01 10:00:00'::timestamptz);
```

`tjsonb(jsonb, tstzset | tstzspan | tstzspanset)` → `tjsonb`. Creates a temporal sequence or a sequence set by associating a constant JSONB value with a temporal domain. The type of the domain determines the resulting structure, either a single sequence or a set of sequences.

```
SELECT tjsonb('{"state":"A"}'::jsonb,
       tstzspan('[2025-01-01 10:00:00, 2025-01-01 12:00:00]'));
```

`tjsonbSeq(tjsonb[], interp [, lowerInc [, upperInc]])` → `tjsonb`. Creates a temporal sequence from an array of `tjsonb` instants. The `interp` parameter controls the temporal semantics (`step` or `discrete`), while `lowerInc` and `upperInc` control the inclusivity of the temporal bounds.

```
SELECT tjsonbSeq(ARRAY[
  '"{\"state\":\"A\"}"@2025-01-01 10:00:00'::tjsonb,
  '"{\"state\":\"B\"}"@2025-01-01 11:00:00'::tjsonb
]);
```

`tjsonbSeqSet(tjsonb[])` → `tjsonb`. Creates a sequence set from an array of `tjsonb` sequences, without modifying their internal structure.

```
SELECT tjsonbSeqSet(ARRAY[
  tjsonb '[{"state":"A"}@2025-01-01 10:00:00]',
  tjsonb '[{"state":"B"}@2025-01-01 12:00:00]'
]);
```

`tjsonbSeqSetGaps(tjsonb[], maxt)` → `tjsonb`. Creates a sequence set from an array of `tjsonb` fragments by introducing explicit discontinuities when the temporal gap between two consecutive elements exceeds `maxt`. If `maxt` is `NULL`, no additional segmentation is applied.

```
SELECT tjsonbSeqSetGaps(ARRAY[
  '"{\"state\":\"A\"}"@2025-01-01 10:00:00'::tjsonb,
  '"{\"state\":\"B\"}"@2025-01-01 12:00:00'::tjsonb
], INTERVAL '30 minutes');
```

A.1.2 Serialization and Exchange Formats

Serialization functions for `tjsonb` provide textual and binary representations suitable for visualization, persistent storage, and data exchange with external systems. These functions follow the same conventions as other MobilityDB temporal types and preserve the temporal semantics of the objects.

`asText(tjsonb)` → `text`. Returns an extended WKT textual representation of a `tjsonb` value. The output is human-readable and can be reused directly as SQL input.

```
SELECT asText(tjsonb '{[{"state":"A"}@2025-01-01 10:00:00]}');
```

`asBinary(tjsonb)` → `bytea`. Serializes a `tjsonb` value into a compact binary WKB representation, suited for efficient storage and transmission.

```
SELECT asBinary( tjsonb '{[{"state":"A"}@2025-01-01 08:00:00]}');
```

`tjsonbFromBinary(bytea)` → `tjsonb`. Reconstructs a `tjsonb` value from its binary WKB representation.

```
SELECT tjsonbFromBinary(
  '\x0142000b010000000100000003120000000000000000010000
  2005000080010000007374617465 4100fcdd5c9ecd0200'
);
```

`asHexWKB(tjsonb)` → `text`. Produces a hexadecimal textual encoding of the WKB binary representation. This format is convenient for text-based storage or interchange.

```
SELECT asHexWKB( tjsonb '{[{"state":"A"}@2025-01-01 08:00:00]}');
```

`tjsonbFromHexWKB(text)` → `tjsonb`. Reconstructs a `tjsonb` value from a hexadecimal WKB string.

```
SELECT tjsonbFromHexWKB(
  '0142000B0100000001000000031200000000000000010000
   2005000080010000007374617465 4100FCDD5C9ECD0200'
);
```

`asMFJSON(tjsonb)` → `json`. Serializes a `tjsonb` value into the MF-JSON (Moving Features JSON) format, intended for interoperability with external services and applications handling temporal data.

```
SELECT asMFJSON( tjsonb '{[{"state":"A"}@2025-01-01 08:00:00]}');
```

`tjsonbFromMFJSON(json)` → `tjsonb`. Reconstructs a `tjsonb` value from its MF-JSON representation.

```
SELECT tjsonbFromMFJSON(
'{
  "type": "MovingJSONB",
  "interpolation": "Step",
  "sequences": [
    {
      "values": [ {"state":"A"} ],
      "datetimes": ["2025-01-01T08:00:00+01"]
    }
  ]
}');
```

A.1.3 Transformation Functions

Transformation functions allow converting a `tjsonb` value between the different temporal subtypes used in MobilityDB (instant, sequence, sequence set) or adjusting its interpolation semantics. These operations preserve both the underlying `jsonb` values and their timestamps, and only affect the temporal structure or interpretation of the object.

`tjsonbInst(tjsonb)` → `tjsonb`. Forces or extracts a `tjsonb` value as a temporal instant. This function is intended for cases where the temporal object represents a single timestamp, even if it is encoded as a sequence or a sequence set.

```
SELECT tjsonbInst( tjsonb '{[{"state":"A"}@2025-01-01 08:00:00]}');
```

`tjsonbSeq(tjsonb)` → `tjsonb`. Converts a `tjsonb` value into a temporal sequence. This normalizes the object into a function defined over a continuous time span, regardless of whether the input was an instant or another temporal representation.

```
SELECT tjsonbSeq( tjsonb '"{\"state\":\"A\"}"@2025-01-01 08:00:00');
```

`tjsonbSeqSet(tjsonb)` → `tjsonb`. Transforms a `tjsonb` value into a sequence set, making temporal discontinuities explicit. This representation is suitable when the temporal domain consists of multiple disjoint intervals.

```
SELECT tjsonbSeqSet( tjsonb '{[{"state":"A"}@2025-01-01 08:00:00]}');
```

`setInterp(tjsonb, text)` → `tjsonb`. Changes the interpolation mode of a `tjsonb` value without modifying its values or timestamps. The specified interpolation applies uniformly to the entire temporal object.

```
SELECT setInterp(tjsonb '{[{"state":"A"}@2025-01-01 08:00:00]}','discrete');
```

A.1.4 Metadata Accessor Functions

Metadata accessor functions allow inspecting the internal structure and properties of a `tjsonb` value without modifying either its contents or its temporal semantics. They are primarily used for introspection, debugging, and query planning, for example to adapt processing logic based on the temporal subtype, interpolation mode, or memory footprint.

`tempSubtype(tjsonb)` → `text`. Returns the temporal subtype of a `tjsonb` value, indicating whether it is internally represented as an *Instant*, a *Sequence*, or a *SequenceSet*.

```
SELECT tempSubtype( tjsonb '"{\"state\":\"A\"}"@2025-01-01 10:00:00');
```

`interp(tjsonb)` → `text`. Returns the interpolation method associated with a `tjsonb` value, such as `Discrete` or `Step`. The interpolation defines how values are interpreted between consecutive instants.

```
SELECT interp(tjsonb '[{"state":"A"}@2025-01-01,{"state":"B"}@2025-01-02]');
```

`memSize(tjsonb)` → `integer`. Returns the amount of memory, in bytes, used to store a `tjsonb` value. This includes both the temporal structure and the embedded `jsonb` payloads.

```
SELECT memSize( tjsonb '"{\"state\":\"A\"}"@2025-01-01 10:00:00');
```

A.1.5 Value Accessor Functions

Value accessor functions extract the `jsonb` payloads carried by a `tjsonb` value, independently of its temporal structure. They operate on the values associated with temporal instants, without modifying or filtering the temporal dimension itself.

`getValue(tjsonb)` → `jsonb`. Returns the `jsonb` value associated with a `tjsonb` of temporal subtype *Instant*.

```
SELECT getValue( tjsonb '"{\"state\":\"A\"}"@2025-01-01 10:00:00');
```

`getValues(tjsonb)` → `setof jsonb`. Returns the set of distinct `jsonb` values carried by a `tjsonb`, regardless of its temporal subtype.

```
SELECT getValues( tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]')
    ;
```

`startValue(tjsonb)` → `jsonb`. Returns the `jsonb` value associated with the first temporal instant of a `tjsonb` object.

```
SELECT startValue( tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]'
    );
```

`endValue(tjsonb)` → `jsonb`. Returns the `jsonb` value associated with the last temporal instant of a `tjsonb` object.

```
SELECT endValue( tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]');
```

`valueN(tjsonb, integer)` → `jsonb`. Returns the `jsonb` value associated with the n-th temporal instant of a `tjsonb`, according to its internal temporal ordering (1-based index).

```
SELECT valueN( tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]',2);
```

A.1.6 Temporal Accessor Functions

Temporal accessor functions extract information related to the temporal dimension of a `tjsonb` value, independently of the `jsonb` payload it carries. They operate exclusively on timestamps, time spans, and durations associated with the temporal function.

`getTimestamp(tjsonb)` → `timestamptz`. Returns the timestamp associated with a `tjsonb` value of temporal subtype *Instant*.

```
SELECT getTimestamp(tjsonb '"{\"state\":\"A\"}"@2025-01-01 10:00:00');
```

`getTime(tjsonb)` → `tstzspanset`. Returns the temporal domain of a `tjsonb` value as a `tstzspanset`, representing the set of time intervals over which the temporal function is defined.

```
SELECT getTime(tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]');
```

`duration(tjsonb[, boundspan])` → `interval`. Returns the total duration during which a `tjsonb` value is defined. When the optional parameter `boundspan` is set to `TRUE`, temporal bounds are explicitly taken into account in the duration computation.

```
SELECT duration(tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-03]');
```

A.1.7 Instant and Timestamp Accessors

Instant and timestamp accessor functions inspect the discrete temporal structure of a `tjsonb` value. They operate on the temporal instants and their associated timestamps, independently of the underlying `jsonb` values.

`numInstants(tjsonb)` → `integer`. Returns the total number of instants composing a `tjsonb` value.

```
SELECT numInstants(
  tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]');
```

`startInstant(tjsonb)` → `tjsonb` **and** `endInstant(tjsonb)` → `tjsonb`. Return respectively the first and the last instant of a `tjsonb` value.

```
SELECT startInstant(
  tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]');
```

`instantN(tjsonb, integer)` → `tjsonb`. Returns the n-th instant of a `tjsonb` value, using 1-based indexing.

```
SELECT instantN(
  tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]',2);
```

`instants(tjsonb)` → `tjsonb[]`. Returns all instants composing a `tjsonb` value as an array of `tjsonb` instants.

```
SELECT instants(
  tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]');
```

`numTimestamps(tjsonb)` → `integer`. Returns the number of distinct timestamps associated with a `tjsonb` value. For discrete temporal subtypes, this equals the number of instants.

```
SELECT numTimestamps(
  tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]');
```

`startTimestamp(tjsonb)` → `timestamptz` **and** `endTimestamp(tjsonb)` → `timestamptz`. Return respectively the first and the last timestamp of a `tjsonb` value.

```
SELECT startTimestamp(
  tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]');
```

`timestampN(tjsonb, integer)` → `timestamptz`. Returns the n-th timestamp associated with a `tjsonb` value.

```
SELECT timestampN(
  tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]',2);
```

`timestamps(tjsonb)` → `timestamptz[]`. Returns all timestamps associated with a `tjsonb` value as an array of `timestamptz`.

```
SELECT timestamps(
  tjsonb '[{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02]');
```

A.1.8 Sequence and Sequence Set Accessors

Sequence and sequence-set accessor functions inspect the higher-level temporal structure of a `tjsonb` value. They operate on the sequences composing the temporal function, in particular when it is defined over multiple disjoint time intervals.

`numSequences(tjsonb)` → `integer`. Returns the total number of sequences composing a `tjsonb` value. For a value defined over a single continuous interval, the result is 1.

```
SELECT numSequences(
  tjsonb '{[
    [{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02],
    [{"state":"C"}@2025-01-04, {"state":"D"}@2025-01-05] ]}');
```

`startSequence(tjsonb)` → `tjsonb` **and** `endSequence(tjsonb)` → `tjsonb`. Return respectively the first and the last sequence of a `tjsonb` value, according to temporal order.

```
SELECT startSequence(
  tjsonb '{[
    [{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02],
    [{"state":"C"}@2025-01-04, {"state":"D"}@2025-01-05] ]}');
```

`sequenceN(tjsonb, integer)` → `tjsonb`. Returns the n-th sequence of a `tjsonb` value, using 1-based indexing.

```
SELECT sequenceN(
  tjsonb '{[
    [{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02],
    [{"state":"C"}@2025-01-04, {"state":"D"}@2025-01-05] ]}', 2);
```

`sequences(tjsonb)` → `tjsonb[]`. Returns all sequences composing a `tjsonb` value as an array of `tjsonb` sequences.

```
SELECT sequences(
  tjsonb '{[
    [{"state":"A"}@2025-01-01, {"state":"B"}@2025-01-02],
    [{"state":"C"}@2025-01-04, {"state":"D"}@2025-01-05]]}');
```

`segments(tjsonb)` → `tjsonb[]`. Decomposes a temporal sequence into elementary segments corresponding to the intervals between consecutive instants. This is useful for fine-grained analysis of temporal transitions within a sequence.

```
SELECT segments(
  tjsonb '[{"state":"A"}@2025-01-01 10:00:00,
          {"state":"B"}@2025-01-01 11:00:00]');
```

A.1.9 Temporal Time-Shift and Scaling Functions

Temporal time-shift and scaling functions transform the temporal domain of a `tjsonb` value without modifying its underlying `jsonb` values. They operate exclusively on timestamps, preserving the temporal structure (instant, sequence, or sequence set) while translating or rescaling time.

`shiftTime(tjsonb, interval)` → `tjsonb`. Translates the entire temporal domain of a `tjsonb` value by a given duration. All timestamps are shifted uniformly by the specified interval.

```
SELECT shiftTime(
  tjsonb '[{"state":"A"}@2025-01-01 10:00:00,
          {"state":"B"}@2025-01-01 11:00:00]',
  INTERVAL '1 hour');
```

`scaleTime(tjsonb, interval)` → `tjsonb`. Applies a temporal scaling factor to the domain of a `tjsonb` value. The time span between instants is proportionally stretched or compressed according to the given interval, while the start time remains fixed.

```
SELECT scaleTime(
  tjsonb '[{"state":"A"}@2025-01-01 10:00:00,
          {"state":"B"}@2025-01-01 11:00:00]',
  INTERVAL '2 hours');
```

`shiftScaleTime(tjsonb, interval, interval)` → `tjsonb`. Combines a temporal shift followed by a temporal scaling. This function allows both repositioning a temporal object in time and modifying its overall duration in a single operation.

```
SELECT shiftScaleTime(
  tjsonb '[{"state":"A"}@2025-01-01 10:00:00,
          {"state":"B"}@2025-01-01 11:00:00]',
  INTERVAL '1 hour',INTERVAL '3 hours');
```

A.1.10 Append and Merge Operations

Append and merge operations allow extending or combining temporal objects of type `tjsonb`. These functions operate on the temporal dimension by adding new instants or sequences, or by merging multiple temporal objects, while preserving the temporal semantics defined by MobilityDB.

`appendInstant(tjsonb, tjsonb)` → `tjsonb`. Appends a single temporal instant to the end of an existing `tjsonb` object. The appended instant must be strictly later than the last instant of the target object. Optional variants allow specifying the interpolation mode used by the resulting temporal value.

```
SELECT appendInstant(
  tjsonb '[{"state":"A"}@2025-01-01 10:00:00]',
  tjsonb '"{\"state\":\"B\"}"@2025-01-01 11:00:00');
```

`appendSequence(tjsonb, tjsonb)` → `tjsonb`. Concatenates two compatible `tjsonb` sequence values in temporal order. This function is used to extend an existing temporal evolution with an additional sequence.

```
SELECT appendSequence(
  tjsonb '[{"state":"A"}@2025-01-01 10:00:00]',
  tjsonb '[{"state":"B"}@2025-01-01 11:00:00]');
```

`merge(tjsonb[, ...])` → `tjsonb`. Combines multiple `tjsonb` objects defined over compatible temporal domains into a single temporal object. Adjacent or contiguous fragments may be coalesced into a single sequence according to temporal semantics.

```
SELECT merge(
  tjsonb '{[{"state":"A"}@2025-01-01 10:00:00]}',
  tjsonb '{[{"state":"B"}@2025-01-01 11:00:00]}');
```

A variant accepts an array of `tjsonb` values and performs a global merge over all elements.

```
SELECT merge(ARRAY[
  tjsonb '{[{"state":"A"}@2025-01-01 10:00:00]}',
  tjsonb '{[{"state":"B"}@2025-01-01 11:00:00]}']);
```

A.1.11 Insertion, Update, and Deletion Operations

Insertion, update, and deletion operations allow modifying an existing `tjsonb` object by adding new temporal values, replacing values over a given temporal domain, or removing parts of its time span. These functions preserve the temporal semantics of MobilityDB and operate exclusively on the temporal structure and associated values.

`insert(tjsonb, tjsonb[, connect])` → `tjsonb`. Inserts a `tjsonb` object into another one, provided that their temporal domains do not overlap. By default, temporally adjacent segments are automatically connected. The optional `connect` parameter can be set to `FALSE` to prevent this automatic merging.

```
SELECT insert(
  tjsonb '{[{"state":"B"}@2025-01-01 11:00:00]}',
  tjsonb '{[{"state":"A"}@2025-01-01 10:00:00]}');
```

`update(tjsonb, tjsonb)` → `tjsonb`. Replaces the values of a `tjsonb` object over the temporal domain covered by another `tjsonb`. This function is typically used to correct or modify existing temporal values.

```
SELECT update(
  tjsonb '{[{"state":"A"}@2025-01-01 10:00:00,
         {"state":"B"}@2025-01-01 11:00:00]}',
  tjsonb '{[{"state":"C"}@2025-01-01 11:00:00]}');
```

`deleteTime(tjsonb, temporal)` → `tjsonb`. Removes a portion of the temporal domain of a `tjsonb` object. The temporal argument may be an instant, a set of instants, a time span, or a set of time spans, allowing flexible removal of single timestamps or entire intervals.

```
SELECT deleteTime(
  tjsonb '{[{"state":"A"}@2025-01-01 10:00:00,{"state":"B"}@2025-01-01
     11:00:00]}',
  timestamptz '2025-01-01 10:00:00');
```

A.1.12 Value-Based Temporal Filtering

Value-based temporal filtering functions allow selecting or excluding portions of a `tjsonb` object according to the `jsonb` values it carries. These operations modify the temporal domain by keeping or removing instants and intervals that satisfy a value-based condition, while preserving the temporal order and structural subtype (instant, sequence, or sequence set).

`atValues(tjsonb, jsonb|jsonbset)` → `tjsonb`. Restricts a `tjsonb` object to the instants or intervals whose value is equal to a given `jsonb` value. A variant accepts a set of values (`jsonbset`) and retains all matching periods.

```
SELECT atValues(
  tjsonb '{[{"state":"A"}@2025-01-01 10:00:00,
           {"state":"B"}@2025-01-01 11:00:00]}','{"state":"A"}'::jsonb);
```

`minusValues(tjsonb, jsonb|jsonbset)` → `tjsonb`. Removes from a `tjsonb` object all temporal components whose value matches the specified `jsonb` value or value set.

```
SELECT minusValues(
  tjsonb '{[{"state":"A"}@2025-01-01 10:00:00,
           {"state":"B"}@2025-01-01 11:00:00]}','{"state":"A"}'::jsonb);
```

`atMin(tjsonb)` → `tjsonb` `minusMin(tjsonb)` → `tjsonb`. The function `atMin` keeps only the periods where the value reaches the global minimum observed in the `tjsonb` object. Conversely, `minusMin` removes these minimal-value periods.

```
SELECT atMin(
  tjsonb '{[{"state":"A"}@2025-01-01,{"state":"B"}@2025-01-02,
           {"state":"A"}@2025-01-03]}');
```

`atMax(tjsonb)` → `tjsonb` `minusMax(tjsonb)` → `tjsonb`. Symmetrically, `atMax` retains the temporal components corresponding to the global maximum value, while `minusMax` excludes them.

```
SELECT atMax(
  tjsonb '{[{"state":"A"}@2025-01-01,{"state":"B"}@2025-01-02,
           {"state":"B"}@2025-01-03]}');
```

A.1.13 Time-Based Temporal Filtering

Time-based temporal filtering functions restrict or exclude portions of a `tjsonb` object according to its temporal domain. These operations act by intersection or subtraction with temporal primitives (`timestamptz`, `tstzset`, `tstzspan`, `tstzspanset`) and preserve the internal structure of the resulting objects (instant, sequence, or sequence set).

`atTime(tjsonb, temporal)` → `tjsonb`. Restricts a `tjsonb` object to the instants or intervals that intersect a given temporal parameter. The temporal argument may be a single timestamp, a set of timestamps, a time span, or a set of spans.

```
SELECT atTime(
  tjsonb '{[{"state":"A"}@2000-01-01,{"state":"B"}@2000-01-02,
           {"state":"C"}@2000-01-03]}',timestamptz '2000-01-02');
```

`minusTime(tjsonb, temporal)` → `tjsonb`. Removes from a `tjsonb` object all temporal components that intersect the specified temporal parameter. The supported temporal types are the same as for `atTime`.

```
SELECT minusTime(
  tjsonb '{[{"state":"A"}@2000-01-01,{"state":"B"}@2000-01-02,
           {"state":"C"}@2000-01-03]}',timestamptz '2000-01-02');
```

`valueAtTimestamp(tjsonb, timestamptz)` → `jsonb`. Returns the `jsonb` value associated with a `tjsonb` object at a specific timestamp. The timestamp must belong to the temporal domain of the object; otherwise, the function returns `NULL`.

```
SELECT valueAtTimestamp(
  tjsonb '{[{"state":"A"}@2000-01-01 10:00:00,
           {"state":"B"}@2000-01-01 11:00:00]}',
  timestamptz '2000-01-01 10:00:00+01');
```

A.1.14 Comparison Functions and B-tree Indexing

Comparison functions define a total ordering over the `tjsonb` type. This ordering is used both for evaluating relational predicates and for enabling efficient indexing with standard PostgreSQL B-tree indexes. Comparisons rely on a canonical internal representation of temporal objects and are consistent across all temporal subtypes (instant, sequence, sequence set).

`temporal_cmp(tjsonb, tjsonb) → integer.` Returns a negative, zero, or positive integer depending on whether the first `tjsonb` value is less than, equal to, or greater than the second one. This function provides the core ordering semantics used internally and by indexes.

Relational comparison operators. The following comparison predicates are defined on pairs of `tjsonb` values and expose the total order through standard SQL operators:

- `<` : strictly less than (`temporal_lt`)
- `<=` : less than or equal (`temporal_le`)
- `=` : equality (`temporal_eq`)
- `<>` : inequality (`temporal_ne`)
- `>=` : greater than or equal (`temporal_ge`)
- `>` : strictly greater than (`temporal_gt`)

These operators are defined uniformly for all `tjsonb` subtypes.

```
SELECT
  tjsonb '"{"state":"A"}"@2000-01-01' =
  tjsonb '"{"state":"A"}"@2000-01-01';
```

```
SELECT
  tjsonb '[{"state":"A"}@2000-01-01]' <
  tjsonb '{[{"state":"A"}@2000-01-01, {"state":"B"}@2000-01-02]}';
```

B-tree indexing. The operator class `tjsonb_btree_ops` enables the use of standard B-tree indexes on columns of type `tjsonb`. Index ordering is based on `temporal_cmp`, allowing efficient evaluation of comparisons, equality predicates, and ORDER BY clauses.

```
CREATE INDEX idx_tjsonb_btree ON my_table USING btree (my_tjsonb_column);
```

A.1.15 Ever/Always Comparison Functions

Ever/always comparison functions evaluate equality or inequality predicates on temporal values by quantifying over their entire temporal domain. They return a single Boolean result that summarizes the temporal evolution of a `tjsonb` object, making them suitable for global temporal conditions and filters.

`ever_eq(jsonb|tjsonb, tjsonb|jsonb) → boolean.` Checks whether equality between the two operands holds at least once over the temporal domain. The result is `TRUE` if there exists at least one instant where the values are equal.

`always_eq(jsonb|tjsonb, tjsonb|jsonb) → boolean.` Checks whether equality holds at all instants of the temporal domain. The result is `TRUE` only if the values are equal for the entire duration.

`ever_ne(jsonb|tjsonb, tjsonb|jsonb) → boolean.` Checks whether inequality holds at least once over the temporal domain.

`always_ne(jsonb|tjsonb, tjsonb|jsonb) → boolean.` Checks whether inequality holds at all instants of the temporal domain.

SQL operators. These functions are exposed through dedicated SQL operators, defined uniformly for the combinations `jsonb-tjsonb`, `tjsonb-jsonb`, and `tjsonb-tjsonb`:

- `?=` : ever equal (`ever_eq`)
- `%=` : always equal (`always_eq`)
- `?<>` : ever not equal (`ever_ne`)
- `%<>`: always not equal (`always_ne`)

```
SELECT tjsonb '"{"state":"A"}"@2000-01-01' ?= jsonb '{"state":"A"}';
```

```
SELECT jsonb '{"state":"A"}'
      %= tjsonb '[{"state":"A"}@2000-01-01, {"state":"A"}@2000-01-02]';
```

A.1.16 Temporal Boolean Operations

Temporal Boolean operations compare `jsonb` and `tjsonb` values while preserving the temporal dimension of the result. Instead of collapsing the comparison into a single Boolean value, these functions return a temporal Boolean (`tbool`) that describes how the predicate evolves over time. They are used when the exact instants or intervals where a condition holds must be retained.

`temporal_teq(jsonb|tjsonb, tjsonb|jsonb) → tbool.` Computes temporal equality between the operands. The result is a `tbool` indicating, at each instant of the temporal domain, whether the values are equal.

`temporal_tne(jsonb|tjsonb, tjsonb|jsonb) → tbool.` Computes temporal inequality between the operands. The resulting `tbool` captures exactly when the values differ over time.

SQL operators. The temporal Boolean functions are exposed through dedicated operators, defined uniformly for `jsonb-tjsonb`, `tjsonb-jsonb`, and `tjsonb-tjsonb` combinations:

- `#=` : temporal equality (`temporal_teq`)
- `#<>` : temporal inequality (`temporal_tne`)

```
SELECT tjsonb '"{"state":"A"}"@2000-01-01' #= jsonb '{"state":"A"}';
```

```
SELECT tjsonb '[{"state":"A"}@2000-01-01,{"state":"B"}@2000-01-02]'
      #<> jsonb '{"state":"A"}';
```

A.1.17 Temporal Relationships with Time Spans

Temporal relationship operators evaluate how the temporal domains of `tjsonb` objects relate to time spans (`tstzspan`) or to other `tjsonb` objects. These operators compare only the temporal extents (instants and intervals), independently of the underlying JSONB values, and return a global Boolean result. They are typically used for temporal joins, filtering, and consistency checks.

`overlaps(tstzspan, tjsonb) → boolean.` Tests whether the temporal domain of a `tjsonb` object overlaps a given time span.

`contains(tjsonb, tstzspan) → boolean.` Checks whether the temporal domain of a `tjsonb` fully contains a given time span.

`contained(tstzspan, tjsonb) → boolean.` Tests whether a time span is fully contained within the temporal domain of a `tjsonb` object.

`same(tjsonb, tjsonb) → boolean.` Checks whether two `tjsonb` objects have exactly the same temporal bounds.

`adjacent(tjsonb, tjsonb) → boolean.` Tests whether the temporal domains of two `tjsonb` objects are adjacent, meaning one ends exactly where the other begins, without overlapping.

SQL operators. These temporal relationship functions are exposed through dedicated operators, defined uniformly for `tstzspan-tjsonb`, `tjsonb-tstzspan`, and `tjsonb-tjsonb` combinations:

- `&&` : temporal overlap (`overlaps`)
- `#@>` : temporal containment (`contains`)
- `<@#` : temporal inclusion (`contained`)
- `~=` : identical temporal bounds (`same`)
- `-|-` : temporal adjacency (`adjacent`)

```
SELECT tstzspan '[2000-01-01,2000-01-03]'
      && tjsonb '[{"state":"A"}@2000-01-01,{"state":"B"}@2000-01-02]';
```

```
SELECT tjsonb '[{"state":"A"}@2000-01-01,{"state":"A"}@2000-01-02]'
      #@> tstzspan '[2000-01-01,2000-01-02]';
```

```
SELECT tstzspan '[2000-01-01,2000-01-02]'
      <@# tjsonb '[{"state":"A"}@2000-01-01, {"state":"B"}@2000-01-03]';
```

```
SELECT tjsonb '[{"state":"A"}@2000-01-01, {"state":"B"}@2000-01-02]'
      ~= tjsonb '[{"state":"C"}@2000-01-01,{"state":"D"}@2000-01-02]';
```

```
SELECT tjsonb '"{"state":"A"}"@2000-01-01'
      -|- tjsonb '"{"state":"B"}"@2000-01-02';
```

A.1.18 Temporal Relative Position Operators

Temporal relative position operators compare the ordering of temporal domains without considering their duration or overlap semantics beyond relative placement. They evaluate whether one temporal object occurs strictly before or after another, or whether it is positioned before/after while allowing partial overlap at the boundary. All operators return a global `boolean` result and apply uniformly to `tstzspan-tjsonb`, `tjsonb-tstzspan`, and `tjsonb-tjsonb` combinations.

`before` (`«#`) → **boolean.** Tests whether the temporal domain of the left operand ends strictly before the temporal domain of the right operand begins.

`overbefore` (`&<#`) → **boolean.** Checks whether the left operand is entirely before the right operand or overlaps it at the beginning.

`after` (`#»`) → **boolean.** Tests whether the temporal domain of the left operand starts strictly after the temporal domain of the right operand ends.

`overafter` (`#&>`) → **boolean.** Checks whether the left operand is entirely after the right operand or overlaps it at the end.

SQL operators. The following operators expose relative temporal position predicates:

- `«#` : strictly before (`before`)
- `&<#` : before or overlapping at the start (`overbefore`)
- `#»` : strictly after (`after`)
- `#&>` : after or overlapping at the end (`overafter`)

```
SELECT tstzspan '[2000-01-01,2000-01-02]'
      <<# tjsonb '[{"state":"A"}@2000-01-03]';
```

```
SELECT tjsonb '[{"state":"A"}@2000-01-01,{"state":"B"}@2000-01-02]'
      &<# tstzspan '[2000-01-02,2000-01-04]';
```

```
SELECT tjsonb '[{"state":"A"}@2000-01-03]'
      #>> tstzspan '[2000-01-01,2000-01-02]';
```

```
SELECT tjsonb '[{"state":"A"}@2000-01-01,{"state":"B"}@2000-01-02]'
      #&> tjsonb '[{"state":"C"}@2000-01-02,{"state":"D"}@2000-01-03]';
```

A.1.19 Aggregate Functions for Temporal JSONB

Aggregate functions for `tjsonb` operate over sets of temporal values, typically produced by SQL queries and groupings. They are used to (i) summarize the temporal extent of multiple objects, (ii) derive temporal aggregates such as counts, and (iii) reconstruct consistent temporal objects from collections of instants or sequences. All functions preserve the temporal semantics defined by MobilityDB.

`extent(tjsonb)` → `tstzspan`. Computes the minimal temporal span covering all aggregated `tjsonb` values. The result spans from the earliest start time to the latest end time observed.

```
SELECT extent(t)
FROM (VALUES
  (tjsonb '[{"state":"A"}@2025-01-01 10:00:00]'),
  (tjsonb '[{"state":"B"}@2025-01-01 12:00:00]')
) AS v(t);
```

`tcount(tjsonb)` → `tint`. Builds a temporal count representing the cumulative number of observed elements over time. The result is a temporal integer (`tint`) aligned with the timestamps of the input values.

```
SELECT tcount(t)
FROM (VALUES
  (tjsonb '"{\"state\":\"A\"}"@2025-01-01 10:00:00'),
  (tjsonb '"{\"state\":\"B\"}"@2025-01-01 11:00:00')
) AS v(t);
```

`merge(tjsonb)` → `tjsonb`. Merges multiple `tjsonb` fragments into a single temporal object. Instants and sequences are ordered in time and coalesced according to standard temporal semantics.

```
SELECT merge(t)
FROM (VALUES
  (tjsonb '"{\"state\":\"A\"}"@2025-01-01 10:00:00'),
  (tjsonb '"{\"state\":\"B\"}"@2025-01-01 11:00:00')
) AS v(t);
```

`appendInstant(tjsonb[, interp[, maxt]])` → `tjsonb`. Constructs one or more temporal sequences from a set of `tjsonb` instants. The optional `interp` parameter controls interpolation, while `maxt` defines the maximum allowed temporal gap before starting a new sequence.

```
SELECT appendInstant(t)
FROM (VALUES
  (tjsonb '"{\"state\":\"A\"}"@2025-01-01 10:00:00'),
  (tjsonb '"{\"state\":\"B\"}"@2025-01-01 11:00:00')
) AS v(t);
```

`appendSequence(tjsonb)` → `tjsonb`. Concatenates already-formed `tjsonb` sequences in temporal order to produce a single continuous temporal trajectory.

```
SELECT appendSequence(s)
FROM (VALUES
  (tjsonb '[{"state":"A"}@2025-01-01 10:00:00]'),
  (tjsonb '[{"state":"B"}@2025-01-01 11:00:00]')
) AS v(s);
```

A.1.20 Temporal Span Extraction and Splitting

Temporal span functions operate on the time domain of a `tjsonb` object. They are used to extract maximal temporal intervals covered by an object and to partition these intervals into regular sub-spans. Such operations are commonly applied in window-based analyses, batching, or temporal segmentation workflows, without altering the underlying JSONB values.

`spans(tjsonb)` → `tstzspan[]`. Returns the set of maximal temporal spans covered by a `tjsonb` object. Each temporal sequence contributes exactly one span corresponding to its temporal extent.

```
SELECT spans(
  tjsonb '{[{"state":"A"}@2025-01-01 10:00:00,
           {"state":"B"}@2025-01-01 12:00:00]}'
);
```

`splitNSpans(tjsonb, integer)` → `tstzspan[]`. Splits the overall temporal domain of a `tjsonb` object into a fixed number of equal-length sub-spans, computed over the full extent of the object.

```
SELECT splitNSpans(
  tjsonb '{[{"state":"A"}@2025-01-01 10:00:00,
           {"state":"B"}@2025-01-01 13:00:00]}', 2);
```

`splitEachNSpans(tjsonb, integer)` → `tstzspan[]`. Applies an equal subdivision independently to each maximal temporal span of a `tjsonb` object. Each sequence is split into the specified number of sub-spans, without interaction with other sequences.

```
SELECT splitEachNSpans(
  tjsonb '{[{"state":"A"}@2025-01-01 08:00:00,
           {"state":"B"}@2025-01-01 09:00:00],
          [{"state":"C"}@2025-01-01 11:00:00,
           {"state":"D"}@2025-01-01 12:00:00]}', 2);
```

A.1.21 Temporal Window and Numeric Projection Functions

These functions apply window-style transformations along the temporal axis or project JSON attributes to numeric temporal types. They are used to derive lagged or lead values in a temporal series and to enable numerical temporal analysis on values originally stored as JSON.

`tjsonb_lag(tjsonb[, int])` → `tjsonb`. Returns a temporally lagged version of a `tjsonb` object. Each instant is assigned the value observed a fixed number of positions earlier in the series, with leading instants yielding `NULL` values.

```
SELECT tjsonb_lag(
  tjsonb '[{"v":1}@2025-03-02 08:00:00,{"v":2}@2025-03-02 09:00:00]');
```

`tjsonb_lead(tjsonb[, int])` → `tjsonb`. The symmetric counterpart of `tjsonb_lag`. Values are shifted forward in time so that each instant receives the value observed later in the temporal sequence.

```
SELECT tjsonb_lead(
  tjsonb '[{"v":10}@2025-04-01 08:00:00,{"v":20}@2025-04-01 09:00:00]');
```

`tjsonb_to_tfloat(tjsonb, text)` → `tfloat`. Extracts a numeric field from the JSON content of each instant and converts it into a temporal floating-point value. This projection enables direct use of numeric temporal analytics such as `twavg`, `ceil`, `floor`, and other arithmetic or statistical operators defined on `tfloat`.

```
SELECT tjsonb_to_tfloat(
  tjsonb '[{"speed":10}@2025-01-01 10:00:00]','speed');
```

A.1.22 JSONB Content Manipulation Functions

These functions operate on the JSONB payload carried by `tjsonb` values. They modify or extract JSON structures at each temporal instant while preserving the temporal dimension. They are typically used to enrich observations with additional attributes, extract specific fields, or update existing JSON content uniformly over time.

`tjsonb_concat(jsonb, tjsonb)` → `tjsonb`
`tjsonb_concat(tjsonb, jsonb)` → `tjsonb`
`tjsonb_concat(tjsonb, tjsonb)` → `tjsonb`. Concatenates JSON objects at each temporal instant. A static `jsonb` argument is merged into the JSON value of every instant, while combining two `tjsonb` values merges their JSON contents instant by instant over their common temporal domain. These functions are exposed via the `||` operator.

```
SELECT '{"state":"A"}'::jsonb ||
      tjsonb '[{"x":1}@2025-01-01 10:00:00,{"x":2}@2025-01-01 11:00:00]';
```

`tjsonb_extract_path(tjsonb, text[])` → `tjsonb`. Extracts, at each instant, the JSON sub-value located at the specified path. The temporal structure of the object is preserved, while the JSON content is reduced to the extracted component.

```
SELECT tjsonb_extract_path(
  tjsonb '[{"speed":10}@2025-01-01 10:00:00]',ARRAY['speed']);
```

`tjsonb_set(tjsonb, text[], jsonb[, boolean])` → `tjsonb`. Updates or inserts a JSON value at the given path for every temporal instant. The optional boolean parameter controls whether missing intermediate keys are created automatically.

```
SELECT tjsonb_set(
  tjsonb '[{"speed":10}@2025-01-01 10:00:00]',
  ARRAY['unit'],'"km/h"'::jsonb);
```

A.2 `jsonbset` Type Reference

A.2.1 Constructors and Input Formats

The `jsonbset` type can be constructed from textual representations or from existing `jsonb` values. Its input syntax follows the same WKT-like conventions as other set types in MobilityDB.

WKT-like text input → jsonbset. A `jsonbset` can be created directly from a textual representation consisting of a set of JSONB values enclosed in braces.

```
SELECT jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }';
```

Construction from `jsonb` values. A singleton set can be created from a single `jsonb` value, or a set can be built from an array of `jsonb` values using the `set` constructor. Equivalent functionality is also available through explicit casts from `jsonb` to `jsonbset`.

```
SELECT set(
  ARRAY['{"state":"A"}'::jsonb, '{"state":"B"}'::jsonb]);
```

A.2.2 Serialization and Exchange Formats

Serialization functions for `jsonbset` provide textual and binary representations suitable for visualization, storage, and data exchange.

`asText(jsonbset)` → `text`. Returns a WKT-like textual representation of a `jsonbset`. The output is human-readable and can be reused directly as SQL input.

```
SELECT asText(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }');
```

`asBinary(jsonbset)` → `bytea`. Serializes a `jsonbset` value into a compact binary WKB representation, suited for efficient storage and transmission.

```
SELECT asBinary(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }');
```

`jsonbsetFromBinary(bytea)` → `jsonbset`. Reconstructs a `jsonbset` value from its binary WKB representation.

```
SELECT jsonbsetFromBinary(
  asBinary(jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }'));
```

`asHexWKB(jsonbset)` → `text`. Produces a hexadecimal textual encoding of the WKB binary representation. This format is convenient for text-based storage or interchange.

```
SELECT asHexWKB(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }');
```

`jsonbsetFromHexWKB(text)` → `jsonbset`. Reconstructs a `jsonbset` value from a hexadecimal WKB string.

```
SELECT jsonbsetFromHexWKB(
  asHexWKB(jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }'));
```

A.2.3 Accessor and Utility Functions

Accessor and utility functions allow inspecting the contents and basic properties of a `jsonbset` value without modifying it. They provide information about memory usage, cardinality, ordering, and enable extraction of individual elements or conversion between set and relational representations.

`memSize(jsonbset)` → `integer`. Returns the amount of memory, in bytes, used to store the `jsonbset` value.

```
SELECT memSize(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }');
```

`numValues(jsonbset)` → `integer`. Returns the number of JSONB elements contained in the set.

```
SELECT numValues(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }');
```

`startValue(jsonbset)` → `jsonb` `endValue(jsonbset)` → `jsonb`. Return respectively the first and the last element of the set, according to the canonical internal ordering.

```
SELECT startValue(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }');
```

`valueN(jsonbset, integer)` → `jsonb`. Returns the n-th element of the set using 1-based indexing.

```
SELECT valueN(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}", "{\"state\":\"C\"}" }',
     2);
```

`getValues(jsonbset)` → `jsonb[]`. Returns all elements of the set as an array of `jsonb` values.

`unnest(jsonbset)` → `setof jsonb`. Expands a `jsonbset` into multiple rows, one per element, enabling direct use in relational queries.

```
SELECT unnest(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }');
```

`setUnion(jsonb|jsonbset)` → `jsonbset`. Aggregate function that builds the union of multiple `jsonb` values or `jsonbset` values into a single `jsonbset`. It is typically used with `GROUP BY` queries to collect distinct JSONB elements.

A.2.4 Comparison Functions and B-tree Indexing

Comparison functions define a total ordering over the `jsonbset` type. This ordering is used both for evaluating relational predicates and for enabling efficient indexing with standard PostgreSQL B-tree indexes. Comparisons rely on a canonical internal representation of `jsonbset` values and are *order-sensitive*: two sets are considered equal only if they contain the same elements in the same order.

`set_cmp(jsonbset, jsonbset)` → `integer`. Returns a negative, zero, or positive integer depending on whether the first `jsonbset` value is less than, equal to, or greater than the second one. This function defines the core ordering semantics and is used internally by all relational operators and by B-tree indexes.

Relational comparison operators. The following comparison predicates are defined on pairs of `jsonbset` values and expose the total order through standard SQL operators:

- `<` : strictly less than (`set_lt`)
- `<=` : less than or equal (`set_le`)
- `=` : equality (`set_eq`)
- `<>` : inequality (`set_ne`)
- `>=` : greater than or equal (`set_ge`)
- `>` : strictly greater than (`set_gt`)

Equality is defined as structural equality: two `jsonbset` values are equal if and only if they contain identical JSON values in the same order. Ordering is lexicographic and based on the ordered sequence of contained `jsonb` elements.

```
-- Comparison of two jsonbset values
SELECT jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }'
    < jsonbset '{ "{\"state\":\"B\"}", "{\"state\":\"C\"}" }';
```

B-tree indexing. The operator class `jsonbset_btree_ops` enables the use of standard PostgreSQL B-tree indexes on columns of type `jsonbset`. Index ordering is based on `set_cmp`, allowing efficient evaluation of equality predicates, range queries, and `ORDER BY` clauses.

```
CREATE INDEX idx_jsonbset_btree
ON test_jsonbset_index
USING btree (traj jsonbset_btree_ops);
```

Queries involving comparison operators can transparently benefit from this index:

```
SELECT *
FROM test_jsonbset_index
WHERE traj <= jsonbset '{ "{\"state\":\"C\"}" }'
ORDER BY traj;
```

A.2.5 Hashing Functions

Hashing functions provide deterministic hash values for the `jsonbset` type and enable its use in hash-based operations such as hash joins, hash aggregation, and hash indexes. Hashing semantics are consistent with equality: two `jsonbset` values that are equal according to `set_eq` always produce identical hash values when the same hashing function and seed are used.

`set_hash(jsonbset)` → `integer`. Computes a 32-bit hash value for a `jsonbset`. The hash is derived from the canonical internal representation of the set and is order-sensitive: sets with the same elements in different orders generally yield different hash values. This function is primarily intended for hash indexes and hash-based grouping.

```
SELECT set_hash(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }');
```

`set_hash_extended(jsonbset, bigint)` → `bigint`. Computes a 64-bit hash value using an explicit seed. The additional seed parameter allows the hash value to be varied deterministically and is required by PostgreSQL for hash joins and other internal hashing mechanisms. For a fixed seed, the function is deterministic and consistent with equality.

```
SELECT set_hash_extended(
  jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }', 42);
```

Hash indexing. The operator class `jsonbset_hash_ops` enables hash indexes on columns of type `jsonbset`. Hash indexes support equality-based lookups and grouping and rely internally on the hashing functions defined above.

```
CREATE INDEX idx_jsonbset_hash
ON my_table
USING hash (my_jsonbset_column);
```

A.2.6 Set Containment and Overlap Operations

Set containment and overlap operations define membership and inclusion relationships over the `jsonbset` type. These predicates operate on the values contained in a set and return a Boolean result. Semantics are value-based and order-insensitive: only the presence or absence of elements is considered, independently of their position in the set.

SQL operators. The containment and overlap predicates are exposed through the following SQL operators:

- `@>` : contains (set–value or set–set)
- `<@` : is contained in (value–set or set–set)
- `&&` : overlap (set–set)

`set_contains(jsonbset, jsonb|jsonbset)` → `boolean`. Tests whether a `jsonbset` contains a given element. The second argument may be either a single `jsonb` value or another `jsonbset`. In the first case, the function checks membership of the value in the set; in the second case, it checks whether all elements of the second set are contained in the first one.

```
SELECT jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }'
      @> jsonb '{"state":"A"}';
```

`set_contained(jsonb|jsonbset, jsonbset)` → `boolean`. Tests whether a value or a set is fully contained in a `jsonbset`. This function is the converse of `set_contains` and supports both value–set and set–set containment checks.

```
SELECT jsonb '{"state":"A"}'
      <@ jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }';
```

`set_overlaps(jsonbset, jsonbset)` → `boolean`. Tests whether two `jsonbset` values share at least one common element.

```
SELECT jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }'
      && jsonbset '{ "{\"state\":\"B\"}", "{\"state\":\"C\"}" }';
```

A.2.7 Relative Position Operators

Relative position operators compare two `jsonbset` values based on their overall ordering. These predicates are derived from the total order defined by the comparison functions and are primarily intended for use with B-tree indexes.

SQL operators. These predicates are exposed through dedicated SQL operators:

- `«` : strictly to the left (`set_left`)
- `»` : strictly to the right (`set_right`)

`set_left(jsonbset, jsonbset) → boolean.` Tests whether all elements of the first `jsonbset` are strictly less than all elements of the second `jsonbset`, according to the canonical lexicographic ordering.

```
SELECT jsonbset '{ "{\"state\":\"A\"}" }'
      << jsonbset '{ "{\"state\":\"C\"}" }';
```

`set_right(jsonbset, jsonbset) → boolean.` Tests whether all elements of the first `jsonbset` are strictly greater than all elements of the second `jsonbset`.

```
SELECT jsonbset '{ "{\"state\":\"C\"}" }'
      >> jsonbset '{ "{\"state\":\"A\"}" }';
```

A.2.8 Set Algebra Operations

Set algebra operations provide union, intersection, and difference semantics for `jsonbset` values. These operations are defined in a value-based manner and ignore ordering: results contain each distinct element at most once, stored in canonical order. All operators support multiple input combinations, allowing set–set, set–value, and value–set variants as appropriate. Operator forms are provided for concise SQL expressions.

`set_union(jsonb|jsonbset, jsonb|jsonbset) → jsonbset.` Returns the union of two inputs, containing all elements that appear in either argument. Variants accept a single `jsonb` value or a `jsonbset` on either side.

```
SELECT jsonbset '{ "{\"state\":\"A\"}" }'
    + jsonbset '{ "{\"state\":\"B\"}" }';
```

`set_intersection(jsonb|jsonbset, jsonb|jsonbset) → jsonbset.` Returns the intersection of two inputs, containing only elements that appear in both arguments. Variants support value–set and set–set combinations.

```
SELECT jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }'
    * jsonbset '{ "{\"state\":\"B\"}", "{\"state\":\"C\"}" }';
```

`set_minus(jsonb|jsonbset, jsonb|jsonbset) → jsonbset.` Returns the difference of two inputs, containing the elements of the first argument that do not appear in the second one. Variants allow subtraction of a single value from a set, a set from a value, or a set from another set.

```
SELECT jsonbset '{ "{\"state\":\"A\"}", "{\"state\":\"B\"}" }'
    - jsonbset '{ "{\"state\":\"B\"}" }';
```

Appendix B

Benchmark Queries

This appendix presents the SQL benchmark queries used in the evaluation.

F1 — Temporal Window Query

MobilityDB

```
SELECT sensor_id
FROM readings_mobility_small
WHERE traj && tstzspan('[2004-03-01, 2004-03-02]');
```

TimescaleDB

```
SELECT DISTINCT sensor_id
FROM readings_timescale_small
WHERE ts BETWEEN '2004-03-01' AND '2004-03-02';
```

F2 — Attribute Threshold Query

MobilityDB

```
SELECT sensor_id
FROM readings_mobility_small
WHERE tfloat(traj, 'temperature') ?> 30;
```

TimescaleDB

```
SELECT DISTINCT sensor_id
FROM readings_timescale_small
WHERE (data->>'temperature')::float > 30;
```

F3 — Time and Threshold Query

MobilityDB

```
SELECT sensor_id
FROM readings_mobility_small
WHERE tfloat(
    atTime(traj, tstzspan('[2004-03-01, 2004-03-02]')),
    'temperature'
) ?> 30;
```

TimescaleDB

```
SELECT DISTINCT sensor_id
FROM readings_timescale_small
WHERE ts BETWEEN '2004-03-01' AND '2004-03-02'
  AND (data->>'temperature')::float > 30;
```

F4 — Count per Sensor

MobilityDB

```
SELECT sensor_id, numInstants(traj)
FROM readings_mobility_small
ORDER BY sensor_id;
```

TimescaleDB

```
SELECT sensor_id, COUNT(*)
FROM readings_timescale_small
GROUP BY sensor_id
ORDER BY sensor_id;
```

F5 — Duration Query

MobilityDB

```
SELECT sensor_id, duration(traj, TRUE)
FROM readings_mobility_small
ORDER BY sensor_id;
```

TimescaleDB

```
SELECT sensor_id, MAX(ts) - MIN(ts)
FROM readings_timescale_small
GROUP BY sensor_id
ORDER BY sensor_id;
```

F6 — Global Time Range

MobilityDB

```
SELECT lower(extent(traj)), upper(extent(traj))
FROM readings_mobility_small;
```

TimescaleDB

```
SELECT MIN(ts), MAX(ts)
FROM readings_timescale_small;
```

F7 — Latest Value

MobilityDB

```
SELECT sensor_id,
       (endValue(traj)->>'temperature')::float
FROM readings_mobility_small
ORDER BY sensor_id;
```

TimescaleDB

```
SELECT sensor_id,
       last((data->>'temperature')::float, ts)
FROM readings_timescale_small
GROUP BY sensor_id
ORDER BY sensor_id;
```

F8 — First Value

MobilityDB

```
SELECT sensor_id,
       (startValue(traj)->>'temperature')::float
FROM readings_mobility_small
ORDER BY sensor_id;
```

TimescaleDB

```
SELECT DISTINCT ON (sensor_id)
       sensor_id,
       (data->>'temperature')::float
FROM readings_timescale_small
ORDER BY sensor_id, ts ASC;
```

F9 — Average Humidity

MobilityDB

```
SELECT sensor_id,
       ROUND(twAvg(tfloat(traj, 'humidity'))::numeric, 6)
FROM readings_mobility_small
ORDER BY sensor_id;
```

TimescaleDB

```
SELECT sensor_id,
       ROUND(AVG((data->>'humidity')::float)::numeric, 6)
FROM readings_timescale_small
GROUP BY sensor_id
ORDER BY sensor_id;
```

F10 — Window Average Temperature

MobilityDB

```
SELECT sensor_id,
       ROUND(
           twAvg(
               tfloat(
                   atTime(
                       traj,
                       tstzspan('[2004-03-01, 2004-03-02]')
                   ),
                   'temperature'
               )
           )::numeric,
           6
       )
FROM readings_mobility_small
ORDER BY sensor_id;
```

TimescaleDB

```
SELECT sensor_id,
       ROUND(AVG((data->>'temperature')::float)::numeric, 6)
FROM readings_timescale_{size}
WHERE ts BETWEEN '2004-03-01' AND '2004-03-02'
GROUP BY sensor_id
ORDER BY sensor_id;
```