Informatics

Flexibles und Effizientes Abfragen von Zeitreihendaten

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Raffael Foidl, BSc
Matrikelnummer 11775820

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dr.techn. Mantas Šimkus, MSc
Mitwirkung: Dipl.-Ing. Dr.techn. Benjamin Mörzinger

Wien, 29. September 2022

Raffael Foidl [illegible] — Mantas Šimkus [illegible]

Raffael Foidl

Mantas Šimkus

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Flexible and Efficient Querying of Time Series Data

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Raffael Foidl, BSc
Registration Number 11775820

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dr.techn. Mantas Šimkus, MSc
Assistance: Dipl.-Ing. Dr.techn. Benjamin Mörzinger

Vienna, 29th September, 2022

Raffael Foidl [illegible]

Mantas Šimkus

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Raffael Foidl, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. September 2022

Raffael Foidl

Raffael Foidl

Danksagung

Ich möchte meinem Betreuer Mantas Šimkus für die kontinuierliche Unterstützung während des gesamten Prozesses dieser Diplomarbeit danken. Sein wissenschaftlicher Rat hat mir nicht nur geholfen, den Rahmen meiner Arbeit abzustecken und einzuhalten, sondern sein wertvolles Feedback hat auch dazu beigetragen, ihren Inhalt auf verständliche Weise zu präsentieren.

Des Weiteren möchte ich mich beim gesamten nista.io-Team für die stets offene und freundlich-lockere Zusammenarbeit bedanken. Vor allem Benjamin Mörzinger und Markus Hoffmann haben entscheidend dazu beigetragen, dass die Beiträge dieser Arbeit ihren domänenspezifischen Anforderungen gerecht werden.

Acknowledgements

I would like to thank my advisor Mantas Šimkus for his continuous guidance throughout the process of this diploma thesis. Not only did his scientific advice help me define and adhere to a viable scope for my work, but his valuable feedback also contributed to me being able to present its contents in an understandable way.

Furthermore, I want to express my gratitude towards the entire team at nista.io for their open and jovial collaboration. Most notably, Benjamin Mörzinger and Markus Hoffmann have played a pivotal part in ensuring that the contributions of this thesis are in accordance with their domain-driven requirements.

Kurzfassung

Das Sammeln großer Mengen von Sensordaten, die von Maschinen erfasst werden—z. B. in Produktionsanlagen—, ist mittlerweile allgegenwärtig. Diese Sensordaten verknüpfen Werte mit der Zeit, zu der sie gemessen wurden, weshalb sie oft auch als Zeitreihendaten bezeichnet werden. Durch die Analyse solcher Aufzeichnungen, die das Verhalten und den Energieverbrauch von Maschinen detailliert beschreiben, können Ineffizienzen und Anomalien aufgedeckt und anschließend zumindest abgeschwächt werden.

Um diese Vorteile von Zeitreihendaten nutzen zu können, sind Mittel zum effizienten Speichern sowie Abfragen ebendieser erforderlich. Es bestehen zwar speziell entwickelte Zeitreihen-Datenbanken, die leistungsstarkes Abspeichern und Verwalten von Zeitreihendaten unterstützen, doch sind ihre nativen Abfrage-Fähigkeiten im Allgemeinen eher grundlegender Natur und operieren auf niedrigem Abstraktionsniveau. Es existieren auch leistungsfähigere Ansätze zur Abfrage von Zeitreihendaten, die unabhängig von konkreten Datenbanken sind. Diese sind jedoch oft konzeptionell komplex und es fehlt ihnen an Werkzeug-Unterstützung, wodurch sie für den Einsatz in industriellen Umfeldern ungeeignet sind.

Deshalb wird in dieser Arbeit DTSQL vorgestellt, eine neue deklarative Zeitreihen-Abfragesprache, deren Kernfunktionalitäten in Zusammenarbeit mit auf Energieeffizienz spezialisierten Domänen-Experten identifiziert wurden. Sowohl ihre Syntax als auch ihre Semantik wurden auf präzise Weise formal definiert und operieren auf hohem Abstraktionsniveau, was es Domänen-Experten ermöglicht, zielgerichtete Abfragen zu formulieren. Darüber hinaus ist sie insofern generisch, als sie unabhängig von spezifischen Zeitreihen-Datenbanken ist. Konkret ist es damit möglich, die Vorteile einer Datenbank zu nutzen und gleichzeitig, mithilfe einer klar definierten Schnittstelle für Datenbankzugriffe, die vorgestellte Abfragesprache zu verwenden.

Sowohl die Spezifikation der vorgeschlagenen Abfragesprache als auch ihre Referenzimplementierung wurden danach bewertet, wie effizient und akkurat sie die während des Anforderungserfassungsprozesses identifizierten Anwendungsfälle abdecken. Die Ergebnisse waren sowohl für die Sprachspezifikation als auch für den implementierten Prototypen recht positiv. Abfragen konnten innerhalb von akzeptablen Zeitspannen ausgewertet werden und ihre Ergebnisse stimmten annähernd mit der von Menschen durchgeführten Bewertung derselben Eingabe-Zeitreihe überein.

Abstract

The practice of collecting large volumes of sensor data captured by machines—e.g., in production plants—has become ubiquitous. These sensor data link values with the time they were measured, which is why they are often also referred to as time series data. By analyzing such records detailing the behavior and energy consumption of machines, inefficiencies as well as anomalies can be detected and subsequently mitigated.

In order to derive these benefits from time series data, efficient means of storing and querying them are required. While there are specifically designed temporal databases supporting high-performance ingestion and storage of time series data, their native query capabilities are generally rather basic and operate on a low abstraction level. There exist more powerful approaches to querying time series data, independent of concrete temporal databases. They often are, however, conceptually rather complex, lack tool support and therefore, are not suitable for use in industrial environments.

Therefore, this thesis proposes `DTSQL`, a novel declarative time series query language whose core features have been identified in collaboration with domain experts specializing in energy efficiency. Its syntax and semantics were precisely and formally defined and provide a high level of abstraction, allowing domain experts to formulate target-oriented queries. Furthermore, it is generic in the sense that it is agnostic to specific temporal databases. More specifically, it possible to leverage the efficiency of a database while also utilizing the proposed query language by means of a predefined interface responsible for storage accesses.

The proposed query language specification and its reference implementation were evaluated according to how efficiently and accurately they cover the use cases identified during the requirement collection process. The results for both the language specification and the implemented prototype system were fairly positive. Query results were obtained within an acceptable amount of time and approximately coincided with the assessment of the same input time series conducted by humans.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

With concepts such as the *Internet of Things* and the *Fourth Industrial Revolution* (Industry 4.0) on the rise, operators of manufacturing plants have gained the opportunity to optimize their production processes based on data captured by their facilities. Sensors and other instruments continuously produce data—more specifically, *time series data*. They comprise a series of values which are associated with the date and time they were measured. If utilized correctly, time series data have the potential to increase productivity, decrease inefficiencies and mitigate anomalies.

High energy consumption leads to high costs. Therefore, it is crucial to be able to locate, analyze and eliminate (or at least reduce) sources of unnecessary energy usage. Conducting analyses of captured time series may help decrease energy expenditure and potentially reduce wear on production machines. Being more energy-efficient is also desirable from an environmental perspective because it entails a decline in CO2 emissions.

In order to derive this desired benefit from time series data, there need to be mechanisms that enable them to be stored and queried efficiently. A mature, established and reliable approach to storing, managing and querying data are *relational database management systems*. They are a good solution to problems with structured data which can be organized in tables. However, they might not be the most appropriate tool for dealing with time series data. This is mainly due to their lack of structure (schemas) as well as their high volatility (noise).

Therefore, dedicated *time series databases* have been built which are designed to support exactly this use case. However, their query languages often operate on a low level of abstraction or are associated with a steep learning curve. Yet, such databases might still be used as baseline for other solutions to build upon them. One could, for example, leverage their efficient storage capabilities as well as utilize them for basic queries such as filtering.

The overall aim of this thesis is to design, implement and formally define the syntax and semantics of a *domain-specific language* (DSL) which counters these issues of existing time series query languages. It should provide an intuitive syntax with a low entry barrier and represent a high-level approach to describing and querying time series data in a way that supports domain experts' practice-oriented requirements. This systematic, generic approach to querying time series data is in direct opposition to repeatedly having to develop individual solutions tailored to a specific use case.

1.2 Research Questions

The preceding section has introduced the broad research area. The concrete goals of this thesis are narrowed down in the following research questions:

Q1) What are the most common, project-agnostic, characteristics domain experts pay attention to when analyzing time series data?

Q2) What language features are required for a time series query language to be expressive enough to capture these characteristics in a declarative way?

Q3) How can the DSL specification be implemented such that the resulting system is able to efficiently evaluate practice-oriented queries?

Q4) To what extent does the prototypical DSL implementation support domain experts in characterizing and finding key features of a given time series?

1.3 Solution Concept

The contribution of this work comprises two core tasks: (i) to *conceptualize and implement a DSL* with a solid formal foundation which is powerful enough to express queries identifying features to be investigated when analyzing time series data. In the following, this language will be referred to as DTSQL (*Declarative Time Series Query Language*). A reference implementation of DTSQL's specification will exemplify its general applicability. This prototype will deal with parsing, interpreting and evaluating queries expressed in DTSQL. Moreover, (ii) a *client environment* will be provided that guides users in constructing and executing valid queries. This relieves them of memorizing the exact syntax and other intricacies usually arising when learning a computer language.

A central requirement of this work is intuitiveness and user assistance during query formulation. An important part of achieving this is the employment of a *structure editor*, also known as *projectional editor*. This means the user is not editing a plain-text query file, but rather operates on a well-defined representation (a projection) of an *abstract syntax* *tree* (AST) of a query, before it is transformed into a textual query. In other words, the editor offers a simplified textual or (semi-)visual way of building the query. Furthermore, it displays context-aware hints and actions such as code completion, intentions (*quick-fixes*), refactoring suggestions and clear, domain-specific error messages.

The requirements and use cases DTSQL should be able to express are acquired in collaboration with external domain experts who work with time series data on a daily basis. Their company offers consulting to other businesses with the goal of helping them achieve a higher level of energy efficiency. This thesis aims to provide them with means of formulating target-oriented queries over time series data. They should be able to retrieve noteworthy features of a given time series without needing to involve computer scientists or database engineers in the process. This leads to higher efficiency *and* efficacy due to less communication overhead as well as a reduced potential of misunderstandings.

An example of what can be expressed using DTSQL is demonstrated in Listing 1.1. This query exhibits the most important language features in the concrete syntax implemented as part of the prototype. At first, it declares two aggregate values (*samples*): the arithmetic mean of the whole data set as well as the minimum value only considering data points between May 28th 2022 at 14:15 and the end of the records. Subsequently, it filters out data points which were measured before the very same date and time. Then, two kinds of periods are characterized (*events*): ones with values consistently above the global average for more than thirty seconds and those with values that are all within a ±25 % range of the local minimum for at least two minutes. Finally, the query *selects* composite periods capturing binary event sequences where, within less than four seconds, a period corresponding to the first event *follows* (occurs after) a period corresponding to the second event. These composite periods are then returned (*yielded*) as query result.

```
WITH SAMPLES:
  avg() AS globalArithmeticMean,
  min("2022-05-28T14:15:00Z", "") AS localMinimum
APPLY FILTER:
  AND(NOT(before("2022-05-28T14:15:00Z")))
USING EVENTS:
  AND(gt(globalArithmeticMean)) FOR (30,] seconds AS aboveAvg,
  AND(around(rel, localMinimum, 25)) FOR [2,] minutes AS aroundMin
SELECT PERIODS:
  (aboveAvg follows aroundMin WITHIN [,4) seconds)
YIELD:
  all periods
```

Listing 1.1: Exemplary DTSQL Query

The source code associated with the reference implementation, its evaluation and the client environment is structured into multiple repositories. They are all hosted on GitHub and available at `https://github.com/dtsql-oss/`.

1.4 Methodology

The thesis encompasses the following research methods:

1. **Literature Review and Research**
   In order to acquire an understanding of the state of the art in the selected problem domain, a review of literature relevant to the topic at hand is conducted. The goal of this is to gain insight into the ways similar problems have been approached and solved. This also enables the identification of research gaps which could be addressed in the thesis.

2. **Requirements Analysis for the Query Language**
   Understanding the requirements and potential use cases of the query language is imperative for designing it adequately. The requirements and desired characteristics are determined as a result of repeated consultations with external domain experts who are working with time series data as part of their occupation in the industry. They have the expertise and experience to explain the use cases and capabilities DTSQL should cover.

3. **Grammar Definition and Language Parsing**
   Based on the requirements and desired language features determined during the requirement collection process, a machine-readable specification of the language syntax needs to be created. This is done under utilization of ANTLR4, i.e., the fourth version of the well-established parser generator. From the grammar definition in ANTLR syntax, the tool automatically generates a lexer and parser.

4. **Formal and Mathematical Modelling**
   Apart from the syntax, the query language's semantics need to be formally defined as well. All operators and language constructs need to have a clearly specified meaning in order to make reasoning about and working with queries on a conceptual level possible. Some language features require the incorporation of logical and mathematical notions in order to express their semantic meaning. For instance, integral or aggregation operators have to involve the corresponding mathematical concepts for a sufficiently precise definition of their semantics.

5. **Implementation of a Prototype**
   As a result of the definition of the query language's syntax and semantics—as elaborated with domain experts according to their requirements—a prototypical implementation of the query language is developed and exposed via a web service. Furthermore, a way for users to formulate and execute queries over their instances of time series data will be provided in the form of a projectional editor. The system is generic in the sense that it is not tightly coupled to a specific storage solution. Instead, it is extensible so that one can add any data source (e.g., a time series database or a custom data provider) to execute queries over by implementing the interfaces provided for this purpose.

6. **Performance Assessment and Evaluation by Domain Experts**
Continuous consultations with domain experts providing feedback from a practice-oriented point of view guides the development of the prototype as well as keeps it on the right track. The prototype's performance will be evaluated quantitatively by repeatedly measuring its runtime with increasing workloads. These findings will be used to locate optimization potentials in the implementation. Furthermore, the system will be evaluated qualitatively by discussing its capabilities and the results it obtained with respect to the requirements and assessment of the domain experts.

1.5 Structure

The rest of the thesis is structured into five main chapters, followed by a final concluding one. The enumeration below outlines their respective purpose.

- **Chapter 2 (Preliminaries)**: This chapter first introduces notable time series databases and describes previous research efforts that have been made towards describing, detecting and querying notable events within given time series data. Furthermore, it explains the notion of domain-specific languages, their development and how they are used in practice—e.g., with projectional editors. Finally, the chapter introduces formal and mathematical concepts which are required for later chapters.
- **Chapter 3 (Collection of Requirements)**: This chapter describes the process of requirements collection as well as its results. It outlines the fundamental requirements expressed by the domain experts accompanying this thesis. Furthermore, it explains the practical use cases that have been identified and which serve as a specification of features to be covered by DTSQL. Their descriptions contain exemplary data instances and plots visualizing the results expected of queries aiming to answer the respective use cases.
- **Chapter 4 (Query Language Specification)**: This chapter is dedicated to the design of the time series query language developed over the course of this thesis. It presents the formal specification of the language resulting from Chapter 3. This entails a definition of the language's abstract syntax—along with a concrete, parser-friendly grammar—and its semantics, precisely characterizing the result of a given query. Furthermore, it provides numerous examples illustrating the (abstract and concrete) syntax as well as the semantics of DTSQL's features.
- **Chapter 5 (Reference Implementation)**: After having established the specification of the query language in Chapter 4, this chapter describes how the specification has been implemented. It explains the software architecture and goes into detail about how queries are parsed, validated and evaluated. Moreover, this chapter demonstrates how the implemented client environment guides and assists users in formulating and executing queries.

- **Chapter 6 (Evaluation)**: This chapter assesses whether and to what extent the goals of this thesis have been achieved. It verifies if the query language is powerful enough to express the requirements collected in Chapter 3. Additionally, this chapter presents a quantitative and qualitative performance assessment: It explores the progression of the query evaluation runtime with increasing input size, and it compares the results computed by the system to intuitive human perception. Finally, it gauges the perceived increase in utility for domain experts provided by this new language, i.e., the improvement relative to the status quo.
- **Chapter 7 (Conclusion)**: This concluding chapter provides a summary of the key points of the thesis. Furthermore, it gives an outlook to future research aspects that could be conducted on this topic which were not possible to address in this thesis.

CHAPTER 2

Preliminaries

2.1 Management of Temporal Data

2.1.1 Time Series Data

The definition of time series data varies depending on author and discipline. The most prominent commonality is that time series generally describe sequences of observations (values) which are ordered by time. They may be continuous—for example, when recording an electrical signal. However, in most cases, they are discrete and measure values at specific time intervals. If data points consist of only one variable, a time series is called *univariate*. If, on the other hand, multiple variables are observed (simultaneously), then it is *multivariate*. [1, 2]

There can also be made a distinction between *historical* and *streaming* time series data. With historical data, there is a clearly defined start and end time of the observations. In the case of streaming data, however, there is no ending point—new measurements are continuously ingested and processed. [3]

The thesis at hand is solely concerned with univariate, historical time series data. A formal characterization of this concept, which will be used throughout the main sections of this document, is provided in Section 2.4.

2.1.2 Temporal Databases and Extensions

There exist numerous time series databases which offer efficient storing and querying capabilities, tailored to their individual (main) goals. For instance, *tsdb* [4] and *Gorilla* [5] focus on monitoring and *TSDS* [6] prioritizes data analytics. Similarly, *Waldo* [7] is designed to operate in potentially non-trustworthy (cloud) environments and therefore, is very much concerned with security and privacy aspects. Lastly, less research-oriented and more general-purpose time series databases such as *InfluxDB*[1], *QuestDB*[2] or *TimescaleDB*[3] may be integrated into more complex (cloud) environments with, for instance, dedicated data ingestion and external monitoring systems.

Such time series data management systems typically offer high throughput and reasonable performance in diverse environments. Moreover, various benchmarks have been conducted that identify the strengths and weaknesses of the different databases [8, 9, 10]. For instance, InfluxDB exhibits efficient ingestion and querying, but has relatively high CPU requirements. TimescaleDB provides decent performance regarding data ingestion and query evaluation, but has a higher memory consumption.

From a technological standpoint, time series databases may expose their data either via a SQL interface or via different mechanisms. For example, InfluxDB offers the SQL-like language *InfluxQL* as well as, more recently, the functional language *Flux* with slightly different capabilities. Both QuestDB and TimescaleDB directly build upon relational database management systems and thus, enable querying via temporal SQL extensions. Neither QuestDB nor TimescaleDB have published scientific papers detailing their respective extensions. However, *TSQL2* (Temporal Structured Query Language) is noteworthy in this context. It is a consensus-based temporal SQL extension that resulted from research efforts spanning multiple years [11, 12, 13]. As a final example, the *Warp 10*[4] database offers rich support for time series analytics via *WarpScript*, a data flow programming language, as well as *FLoWS*, a functional language whose feature set is equivalent to WarpScript's.

2.2 Event Detection in Time Series Data

There have already been substantial research efforts concerned with representing, querying and reasoning about time series or temporal data in general. Some of these studies are rather abstract and on the theoretical side. Others have more practice-oriented contributions, such as concrete query languages. The following paragraphs give a succinct overview of scientific work related to the representation of and information extraction from time series data.

A lot of focus has been directed to temporal and description logics regarding the management and monitoring of temporal data. For example, [14] maps a subset of TSQL2 to temporal logic and vice versa. Similarly, [15] describes properties of time series data using temporal logic. On the other hand, [16] employs description logics to represent and query temporal data. On top of that, an approach to reasoning about temporal data that combines both temporal and description logics is presented in [17].

[1] `https://www.influxdata.com/`
[2] `https://questdb.io/`
[3] `https://www.timescale.com/`
[4] `https://www.warp10.io/`

There are powerful query languages for time series or sensor log data. Rule-based languages such as *datalogMTL* [18] and *DslD* [19] are powerful enough to characterize events using multiple constraints over both the time and the value dimension. While they are able to cover a wide array of use cases due to their high level of expressiveness, they are often not suitable for widespread use. The main reasons for that are the required level of expertise in formal methods, difficulty of implementation and therefore lacking tool support. The amount of work necessary to formulate and execute purposeful queries is too high for practical purposes.

There have been efforts to create systems that, by design, avoid this by allowing to specify queries visually [20, 21, 22, 23]. These works are, however, not limited to time series data alone. Solutions have also been proposed for two-dimensional data in general [24], *semantic* data (i.e., data with ontological representations) [25] and graphs [26].

The problem of locating trends, specific patterns or shapes in time series data has also been studied extensively by employing formal, mathematical and/or statistical methods [27, 28, 29, 30, 31]. This may entail similarity or distance measures, (piece-wise) linearizations of the input data, probabilistic models, summary statistics, trigonometric functions as well as (discrete) differentiation.

Similarly, there are various studies which are concerned with detecting anomalies in time series data. In this area of research, the goal is not to detect concrete shapes in a time series, but to recognize events which are out of the ordinary. For example, [32] does so under utilization of higher-order finite differences (see also Section 2.6.1). The approach published by [33], on the other hand, employs neural networks to detect anomalies. Moreover, [34] uses a stochastic model in the shape of a dynamic Markov model.

Lastly, Allen's interval algebra [35] also deserves to be mentioned for its importance in theoretical and also practical contributions that are concerned with time intervals. It introduces notions which enable expressing and reasoning about thirteen different temporal relationships between time intervals. They are depicted in Figure 2.1.

As the figure shows, there are seven unique relations, all of which (except *equal*) also define a unique inverse relation. For instance, X *before* Y expresses that the interval X occurs before the interval Y, with X's ending point being strictly before Y's starting point. On the other hand, X *meets* Y has a very similar meaning, except that X's ending point coincides with Y's starting point. As a final example, Y *started-by* X denotes that the starting points of X and Y coincide, while X's ending point is strictly before Y's.

2.3 Domain-Specific Languages

The main objective of this thesis is to design, specify and implement a *domain-specific language* (DSL) with the purpose of facilitating the detection of noteworthy intervals and events in time series data. Therefore, this section gives a brief introduction to DSLs as well as tools and concepts associated with their development.

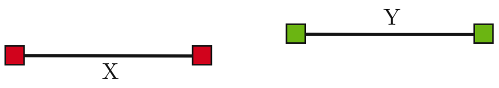
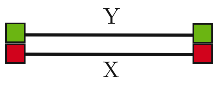
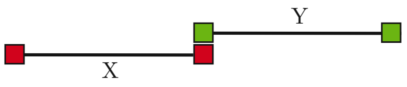
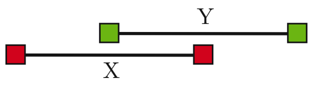
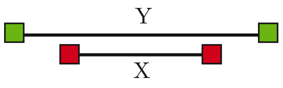
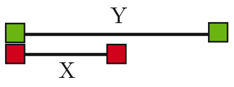
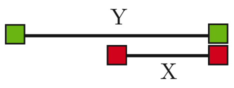
Relation	**Inverse Relation**	**Illustration**
X before Y	*Y after X*	
X equal Y	*Y equal X*	
X meets Y	*Y met-by X*	
X overlaps Y	*Y overlapped-by X*	
X during Y	*Y contains X*	
X starts Y	*Y started-by X*	
X finishes Y	*Y finished-by X*	

Figure 2.1: Allen's Interval Relations (Adapted From [35])

2.3.1 DSLs and Language-Oriented Programming

Even though domain-specific languages have been designed, used and studied for several years now, there is no exact and generally agreed upon definition. In essence, however, they are programming languages with limited expressiveness and are geared towards or restricted to a particular problem domain. Since they usually operate on a deliberately higher level of abstraction than general-purpose programming languages, it is not uncommon for domain-specific languages to be *declarative*. In other words, it enables language users to specify *what* they want to achieve, without requiring them to describe *how* to do so (as it is the case with *imperative* programming languages). [36]

Occasionally, a distinction is made between *external* and *internal* domain-specific languages. According to [37, pp. 27–28], external DSLs are independent languages that have their own syntax as well as parser, interpreter and/or compiler which are developed in another programming language. In contrast to that, internal DSLs use a subset of a general-purpose language such that they feel like a custom language, while still being able to reuse existing tools such as parsers, compilers, or even debuggers.

Adopting DSLs may result in benefits like the ability to express problems concisely and on an abstraction level adequate for the respective domain, with the possibility of an increase in productivity. However, the effort of educating users in (new) DSLs as well as the cost of designing, implementing and maintaining them has also to be taken into account. It is very well possible that, for a given use case, the advantages of introducing a domain-specific language do not outweigh their cost, in comparison to hand-coded (ad hoc) software. [36]

A term that is sometimes brought up in the context of DSLs is the paradigm of *language-oriented programming*. It has as central aspect the development of formally well-defined, use-case-specific, high-level languages. They should be able to express domain-oriented problems succinctly and also capture domain knowledge—e.g., LaTeX encapsulates complex rules about the typsetting of mathematical formulas [38, 39]. In this sense, the thesis at hand embraces language-oriented programming with the formal specification and implementation of DTSQL. It is an external DSL that, by design, captures knowledge about time series analysis and allows for the succinct formulation of common tasks in this domain.

Closely linked to this mentality are *language workbenches*, development environments which help create DSLs as well as design custom editing environments to write DSL scripts [37, p. 28]. The next subsection provides an introduction to language workbenches, their capabilities and also mentions some notable examples.

2.3.2 Language Workbenches

As already mentioned above, language workbenches serve the purpose of facilitating the process of creating DSLs along with associated tools (development environments, parsers, debuggers, etc.). They often provide a level of support to both DSL creators as well as users writing DSL scripts that is comparable to modern development environments for (general-purpose) programming languages [37, pp. 129–130]. This may include assisted textual editing, auto-complete, intentions (*quick-fixes*), refactoring suggestions or even graphical editors.

There exist numerous language workbenches, e.g., the *Meta Programming System*[5] (MPS) [39, 40, 41] by JetBrains, *Xtext*[6] [42], *MetaEdit+*[7] [43], *Spoofax* [44] or *Cedalion* [45]. They all possess individual strengths and weaknesses, which is why various efforts have been made to evaluate available language workbenches. For instance, [46] focuses in on an application of MPS in the domain of requirements specification, and [47] is a study comparing many workbenches, based on an especially established feature model. Moreover, several *language workbench competitions* have been held with the same goal, most notably in 2013 [48] and 2016 [49].

The editors of language workbenches offer users the ability to create and manipulate DSL scripts/programs in a textual, graphical, tabular or *projectional* manner. While graphical editors may provide the ability to edit the underlying structure of a DSL program via a diagram, projectional editors—e.g., as provided by MPS—let users operate on a projection of the DSL's internal model defined by the language designer [47]. More

[5]https://www.jetbrains.com/mps/
[6]https://www.eclipse.org/Xtext/
[7]https://www.metacase.com/products.html

precisely, this projection is a mapping from the *abstract syntax tree* of a DSL program to an editable, designer-defined representation of it [50]. This representation, again, might be textual, (semi-)graphical, tabular or contain special symbols and images. Since projectional editors build upon a DSL's structure, they are also called *structure editors*.

While projectional editors have been around for many years now, they have not been able to establish themselves on a broad scale, neither in the context of language workbenches nor in development environments for general-purpose programming languages. There are multiple studies regarding the acceptance and efficiency of projectional editors [50, 51]. Common findings are that, while fundamentally useful, they require programmers to readjust from their accustomed style of plain-text source code editing, which they are often reluctant to due to insufficient prospective benefit. Furthermore, the implementation of practical, user-friendly projectional editors turns out to be rather difficult.

The scope of this thesis does not allow going into further detail about the anatomy and mechanics of particular language workbenches and their editors. However, [39] contains an explanation of its MPS's design philosophy and [40] an overview of its most important aspects. What is more, an in-depth view of MPS's features with focus on industrial applications, research projects and educational aspects is presented in [52]. Finally, Section 5.3 (Client Environment) explains how MPS was employed in this thesis to create a projectional editor for DTSQL queries.

2.4 Temporal Notions

The concepts introduced in later chapters of this thesis, especially in Chapter 4 (Query Language Specification), are strongly connected with notions related to time. This section provides formal definitions of our representation of time, time series and time intervals so that their meaning is well-understood when they are used throughout the document.

2.4.1 Time

Specific points in time are represented as integers $\mathsf{t} \in \mathbb{Z}$. They are typographically distinguished from "regular" integer variables by means of a sans serif font. Definition 2.1 captures this notion of time formally.

Definition 2.1 (Time as a Total Order) *The concept of time is modelled by the total order $(\mathbb{Z}, \leq)$. Therefore, these properties hold for any arbitrary point in time:*

1. reflexivity*:* $\forall \mathsf{t} \in \mathbb{Z}\colon\ \mathsf{t} \leq \mathsf{t}$
2. reflexivity*:* $\forall \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3 \in \mathbb{Z}\colon\ (\mathsf{t}_1 \leq \mathsf{t}_2 \wedge \mathsf{t}_2 \leq \mathsf{t}_3) \implies \mathsf{t}_1 \leq \mathsf{t}_3$
3. antisymmetry*:* $\forall \mathsf{t}_1, t_2 \in \mathbb{Z}\colon\ (\mathsf{t}_1 \leq \mathsf{t}_2 \wedge \mathsf{t}_2 \leq \mathsf{t}_1) \implies \mathsf{t}_1 = \mathsf{t}_2$
4. totality*:* $\forall \mathsf{t}_1, \mathsf{t}_2 \in \mathbb{Z}\colon\ \mathsf{t}_1 \leq \mathsf{t}_2 \vee \mathsf{t}_2 \leq \mathsf{t}_1$

This definition is reasonable because it is in accord with our intuitive understanding of time. Reflexivity is naturally satisfied because any point in time coincides with itself. Transitivity of any sequence of events is given by the fact that time—at least in its day-to-day understanding—progresses linearly. If two points in time precede or coincide with each other, they must be equal, which corresponds to antisymmetry. Lastly, as for totality, given two points in time, one of them must occur before or coincide with the other (there are no two points which cannot be ordered temporally).

Note that it would also be possible to use the total order constituted by $(\mathbb{N}_{\geq 0}, \geq)$. However, this order exhibits a lower bound, zero. Since—at least in theory—there is no earliest date or point in time, $(\mathbb{Z}, \leq)$ is the better choice of analogy. Furthermore, we do not specify an explicit function projecting time variables $\mathtt{t} \in \mathbb{Z}$ onto specific dates and times because such a mapping is not required for the purpose of this thesis. The way of representing time is implementation-specific and not dictated by the language specification, as long as the validity of the properties stated by Definition 2.1 is preserved.

We do, however, assume the existence of some time-related functions which allow for a higher level of expressiveness and convenience when working with dates and times. They are declared in Definition 2.2—their concrete definition is, again, left to the implementation.

Definition 2.2 (Time-Related Functions)

1. $\mathtt{extract}(\mathtt{t}, c) \in \mathbb{N}$, $\mathtt{t} \in \mathbb{Z}$, $c \in \{\mathtt{year}, \mathtt{month}, \mathtt{day}, \mathtt{hour}, \mathtt{minute}, \mathtt{second}, \mathtt{milli}\}$*:*
   Extracts a time component c *of a given point* $\mathtt{t}$. *The range of the function depends on the choice of* c*:*
   - $\mathtt{extract}(\mathtt{t}, \mathtt{year}) \in \mathbb{N}_{\geq 1}$
   - $\mathtt{extract}(\mathtt{t}, \mathtt{month}) \in \{1, 2, \ldots, 12\}$
   - $\mathtt{extract}(\mathtt{t}, \mathtt{day}) \in \{1, 2, \ldots, 31\}$
   - $\mathtt{extract}(\mathtt{t}, \mathtt{hour}) \in \{0, 1, 2, \ldots, 23\}$
   - $\mathtt{extract}(\mathtt{t}, \mathtt{minute}) \in \{0, 1, 2, \ldots, 59\}$
   - $\mathtt{extract}(\mathtt{t}, \mathtt{second}) \in \{0, 1, 2, \ldots, 59\}$
   - $\mathtt{extract}(\mathtt{t}, \mathtt{milli}) \in \{0, 1, 2, \ldots, 999\}$
2. $\mathtt{duration}(\mathtt{t}_\Delta, u) \in \mathbb{R}$, $\mathtt{t}_\Delta \in \mathbb{N}_{\geq 0}$, $u \in \{\mathtt{weeks}, \mathtt{days}, \mathtt{hours}, \mathtt{minutes}, \mathtt{seconds}, \mathtt{millis}\}$*:*
   Returns the duration represented by the time difference $\mathtt{t}_\Delta$ *between two points in time in the unit* u.

2.4.2 Time Series

One of the central concepts used throughout this work are *time series* because they contain the data that queries are ultimately executed on. Intuitively, we understand a time series as an ordered sequence of time-value pairs, so-called *data points*. Definition 2.3 provides a definition of these two concepts and their most important properties.

Definition 2.3 (Data Points and Time Series) *A* data point *is a pair*

$$p := (\mathsf{t}, v), \textit{ with } \mathsf{t} \in \mathbb{Z},\ v \in \mathbb{R}, \tag{2.1}$$

meaning that at time t*, the value* v *was recorded. The functions* `dpt` *and* `dpv` *return the* time *and* value component *of a data point* p*, respectively, and are defined as*

$$\texttt{dpt}(p) = \texttt{dpt}((\mathsf{t}, v)) := \mathsf{t} \tag{2.2}$$
$$\texttt{dpv}(p) = \texttt{dpv}((\mathsf{t}, v)) := v \tag{2.3}$$

A time series *is a sequence of data points*

$$\Psi = \langle p_k \rangle = \langle (\mathsf{t_k}, v_k) \rangle, \textit{ with } 0 \le k \le n \tag{2.4}$$

that is ordered by the time component of the data points as per $(\mathbb{Z}, \le)$*. The* size *or* length *of a time series is denoted by* $|\Psi|$ *and is equal to the number of entries, i.e.,*

$$|\Psi| := n \tag{2.5}$$

The set of data points contained in a time series Ψ *is denoted by*

$$\mathcal{P}(\Psi) := \{p \mid p \in \Psi\} \tag{2.6}$$

We assume that data points in a time series are unique with respect to their time component, i.e.,

$$\forall p_i, p_k \in \Psi\colon\ i \ne k \implies \mathsf{t_i} \ne \mathsf{t_k} \tag{2.7}$$

The definition above states that data points in a time series are in ascending order and unique, both with respect to time. This further means that the elements of a time series are even *strictly* monotonically increasing with respect to time, i.e.,

$$\forall i, j \in \mathbb{N}_{\ge 0},\ p_i, p_j \in \Psi\colon\ i > j \implies \mathsf{t_i} > \mathsf{t_i} \tag{2.8}$$

holds. This assumption facilitates designing algorithms for the implementation of the language specification significantly. Example 2.1 illustrates the interplay of definitions introduced in this and the previous subsection.

Example 2.1 (Data Points and Time Series) *Consider the (very small) extract of data recorded by a sensor depicted in Table 2.1.*

It defines the time series

$$\Psi = \langle p_1, p_2, p_3 \rangle = \langle (\mathsf{t}_1, 524.5), (\mathsf{t}_2, 530.23), (\mathsf{t}_3, 544.72) \rangle \tag{2.9}$$

timestamp	**identifier**	**value**
`2022/08/16 13:30:45`	t_1	524.50
`2022/08/16 14:00:45`	t_2	530.23
`2022/08/16 14:30:45`	t_3	544.72

Table 2.1: Exemplary Time Series for Example 2.1

with its unordered point set $\mathcal{P}(\Psi) = \{p_1, p_2, p_3\}$ *and size* $|\Psi| = 3$. *All values were recorded in the year*

$$\begin{aligned}\texttt{extract}(\texttt{dpt}(p_1), \texttt{year}) &= \texttt{extract}(\texttt{dpt}(p_2), \texttt{year}) \\ &= \texttt{extract}(\texttt{dpt}(p_3), \texttt{year}) = 2022\end{aligned} \tag{2.10}$$

and the highest value is

$$\texttt{dpv}(p_3) = 544.72 \tag{2.11}$$

Finally, the total duration captured by Table 2.1 is

$$\texttt{duration}(\underbrace{\texttt{dpt}(p_3) - \texttt{dpt}(p_1)}_{t_\Delta}, \texttt{minutes}) = 60 \tag{2.12}$$

minutes. △

2.4.3 Time Intervals

A *time interval*, or short *interval*, of a time series is a segment of time that is defined by a *lower bound* and *upper bound* and captures a subset of said time series. The specification of an interval is formulated more formally in Definition 2.4.

Definition 2.4 (Time Intervals) *Let* Ψ *be a time series,* $\mathsf{t_i} \in \mathbb{Z}$ *a lower bound and* $\mathsf{t_j} \in \mathbb{Z}$ *an upper bound with* $\mathsf{t_j} \geq \mathsf{t_i}$. *Then, the interval*

$${}_i\pi_j \tag{2.13}$$

is representative of all data points from Ψ *that are within the bounds* $\mathsf{t_i}$ *and* $\mathsf{t_j}$. *More specifically, the* point set *of an interval is given by*

$$\texttt{ps}_\Psi({}_i\pi_j) \coloneqq \{p \in \Psi \mid \texttt{dpt}(p) \geq \mathsf{t_i} \wedge \texttt{dpt}(p) \leq \mathsf{t_j}\} \tag{2.14}$$

The functions

$$\begin{aligned}\texttt{intstart}({}_i\pi_j) &\coloneqq \mathsf{t_i} \\ \texttt{intend}({}_i\pi_j) &\coloneqq \mathsf{t_j}\end{aligned} \tag{2.15}$$

denote the lower and upper bounds of $_i\pi_j$*, respectively. Moreover, the* length *or* duration *of an interval is non-negative and expressed by*

$$|_i\pi_j| \coloneqq \texttt{intend}(_i\pi_j) - \texttt{intstart}(_i\pi_j) \tag{2.16}$$

It can be translated to familiar units with the `duration` *function.*

An illustration of how intervals may be used to denote subsection of a concrete time series is depicted in Example 2.2.

Example 2.2 (Time Intervals) *Consider the time series* $\Psi = \langle p_1, \ldots, p_5 \rangle$ *defined by Table 2.2.*

data point	time	identifier	value
p_1	`2022-08-21 17:35:00`	$\mathtt{t}_1$	230.23
p_2	`2022-08-21 17:50:00`	$\mathtt{t}_2$	195.50
p_3	`2022-08-21 18:05:00`	$\mathtt{t}_3$	215.30
p_4	`2022-08-21 18:20:00`	$\mathtt{t}_4$	225.75
p_5	`2022-08-21 18:35:00`	$\mathtt{t}_5$	232.13

Table 2.2: Exemplary Time Series for Example 2.2

The interval $_2\pi_4$ *consists of the data points*

$$\texttt{ps}_\Psi(_2\pi_4) = \{p_2, p_3, p_4\} \tag{2.17}$$

and its bounds are given by

$$\begin{aligned}\texttt{intstart}(_2\pi_4) &= \mathtt{t}_2\\ \texttt{intend}(_2\pi_4) &= \mathtt{t}_4\end{aligned} \tag{2.18}$$

The length of $_2\pi_4$ *is equal to*

$$|_2\pi_4| = \texttt{intend}(_2\pi_4) - \texttt{intstart}(_2\pi_4) = \mathtt{t}_4 - \mathtt{t}_2 \tag{2.19}$$

which is equivalent to

$$\texttt{duration}(|_2\pi_4|, \texttt{hours}) = \texttt{duration}(\mathtt{t}_4 - \mathtt{t}_2, \texttt{hours}) = 0.5 \tag{2.20}$$

hours. △

2.5 Propositional Formulas

The syntax of DTSQL, the temporal query language presented in this thesis, directly incorporates propositional logic—see Section 4.1.3 (Filters) and Section 4.1.4 (Events).

While it is most likely that the reader is familiar with this formalism, the following paragraphs will give a brief overview of its syntax and semantics. This description is kept at a minimum, further definitions and explanations may be found in any standard textbook on (classical) logic, e.g., [53].

In Definition 2.5, we define a syntactically restricted subset of propositional formulas that only exhibits conjunction, disjunction, and negation. This restriction is made in order to keep the syntax of the language proposed in Chapter 4 compact. We do not lose expressiveness by this measure because conjunction and negation—as well as disjunction and negation—are *functionally complete* sets of logical connectives. This means that every other logical operator can be expressed as a combination of the operators in one such set. A proof of this claim is provided in [53, p. 83].

Definition 2.5 (Syntax of Propositional Formulas) *Let* $\mathcal{V} = \{v_1, v_2, \dots\}$ *be a set of* propositional variables*—the terms* atomic formulas *or* (propositional) atoms *are also applicable. Furthermore, let* $\mathbb{B} := \{\texttt{true}, \texttt{false}\}$ *be the set of* propositional constants*. Then, the set* $\mathcal{F}$ *of* propositional formulas *is defined inductively as the smallest set such that*

1. $\mathcal{V} \subseteq \mathcal{F}$

2. $\mathbb{B} \subseteq \mathcal{F}$

3. If $A \in \mathcal{F}$*, then* $\neg A \in \mathcal{F}$*.*

4. If $A, B \in \mathcal{F}$*, then* $(A \circ B) \in \mathcal{F}$*, with* $\circ \in \{\wedge, \vee\}$*.*

Example 2.3 depicts a few exemplary propositional formulas according to the definition used in the thesis at hand.

Example 2.3 (Syntax of Propositional Formulas) *All formulas below are examples of syntactically valid propositional formulas according to Definition 2.5.*

- v_4
- $\neg\texttt{false}$
- $(\neg v_1 \vee v_2)$
- $(\neg(v_3 \wedge \neg v_5) \wedge \neg(v_5 \wedge \neg v_3))$

$\triangle$

The *(propositional) value* or the *result of the evaluation* of a propositional formula f depends on an *interpretation (function)* which assigns boolean values (propositional constants) to atoms. In addition to that, the value of non-constant and non-atomic formulas—i.e., ones constructed by rule 3 or rule 4—is determined using the functions depicted by Table 2.3. These two factors are combined in Definition 2.6, formalizing the semantics of propositional formulas.

i	$\neg i$
`false`	`true`
`true`	`false`

(a) Truth Table of Negation

i_1	i_2	$i_1 \wedge i_2$	$i_1 \vee i_2$
`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`
`true`	`false`	`false`	`true`
`true`	`true`	`true`	`true`

(b) Truth Table of Conjunction and Disjunction

Table 2.3: Truth Table of Logical Operators Supported By Definition 2.5

Definition 2.6 (Semantics of Propositional Formulas) *Let $f \in \mathcal{F}$ be a propositional formula over the set of variables $\mathcal{V}$. Furthermore, let $\mathcal{I} = \{I \mid I\colon \mathcal{V} \to \mathbb{B}\}$ be the set of all interpretation functions mapping atoms to truth values. Then, the value of f* with respect to a concrete interpretation *$I \in \mathcal{I}$ is determined by the* evaluation function *$\texttt{eval}_I\colon \mathcal{I} \times \mathcal{V} \to \mathbb{B}$ which is defined inductively:*

1. *If $v \in \mathcal{V}$, then $\texttt{eval}_I(v) := I(v)$.*
2. $\texttt{eval}_I(\texttt{true}) := \texttt{true}$ *and* $\texttt{eval}_I(\texttt{false}) := \texttt{false}$
3. *If $A \in \mathcal{F}$, then $\texttt{eval}_I(\neg A) := \neg(\texttt{eval}_I(A))$, where the function $\neg\colon \mathbb{B} \to \mathbb{B}$ is defined as in Table 2.3a.*
4. *If $(A \circ B) \in \mathcal{F}$, with $\circ \in \{\wedge, \vee\}$, then $\texttt{eval}_I((A \circ B)) := \circ(\texttt{eval}_I(A), \texttt{eval}_I(B))$, where the function $\circ\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ is defined as in Table 2.3b.*

To conclude this section, Example 2.4 demonstrates a step-by-step evaluation of a propositional formula.

Example 2.4 (Semantics of Propositional Formulas) *Let $\mathcal{V} = \{v_1, v_2, v_3\}$ be a set of atomic formulas, $f = (\neg(v_1 \vee v_2) \wedge v_3)$ a propositional formula over $\mathcal{V}$ and*

$$I\colon \mathcal{V} \to \mathbb{B},\ v \mapsto \begin{cases} \texttt{false}, & \text{if } v = v_1 \text{ or } v = v_2 \\ \texttt{true}, & \text{if } v = v_3 \end{cases} \tag{2.21}$$

an interpretation function. Then, the value of f is determined by $\texttt{eval}_I$ *as follows:*

$$\begin{aligned}\texttt{eval}_I(f) &= \texttt{eval}_I((\neg(v_1 \lor v_2) \land v_3)) \\ &= \land(\texttt{eval}_I(\neg(v_1 \lor v_2)), \texttt{eval}_I(v_3)) \\ &= \land(\neg(\texttt{eval}_I(v_1 \lor v_2)), \texttt{true}) \\ &= \land(\neg(\lor(\texttt{eval}_I(v_1), \texttt{eval}_I(v_2))), \texttt{true}) \\ &= \land(\neg(\lor(\texttt{false}, \texttt{false})), \texttt{true}) \\ &= \land(\neg(\texttt{false}), \texttt{true}) \\ &= \land(\texttt{true}, \texttt{true}) \\ &= \texttt{true}\end{aligned} \tag{2.22}$$

△

2.6 Mathematical Concepts

Defining the semantics of DTSQL also requires some mathematical concepts. This section provides a brief introduction to three methods which will be involved in Section 4.2 (Language Semantics)—numerical differentiation, numerical integration and linear regression. The explanations of these methods, as presented in this section, are in line with how they are commonly defined. While they may be found in any standard textbook related to the respective subject, initial resources for further reading will be provided.

2.6.1 Numerical Differentiation

This subsection has been adapted from [54, pp. 38–44] and [55, pp. 529–544]. Refer to them for more detailed elaborations.

When talking about the rate of change of a function—e.g., when examining whether the values are on an increasing or decreasing trend—we resort to the concept of differentiation. Geometrically speaking, the derivative f' of a continuous function f at x is the *slope of the tangent line* at x and defined as

$$f'(x_0) \coloneqq \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} \tag{2.23}$$

Intuitively, this means that the slope of the tangent line is the result of a process where the difference between two points defining a secant line has become infinitely small.

This thesis primarily works with time series data consisting of discrete measurements. Their smallest possible Δx is dictated by the resolution of the data, i.e., the sampling rate of the sensor recording the time series. Hence, it is not feasible to actually calculate the limit as in Equation (2.23). We have to approximate the derivative of a time series by the slope of a secant line.

This method of determining a numerical derivative using positive offsets defined by $+\Delta x$ is known as *forward finite difference approximation*. Similarly, considering negative steps

$-\Delta x$ is called a *backward difference*. The combination of both uses the average of forward and backward differences and is referred to as the *central difference*. Using these notions, Definition 2.7 depicts three approximations of a derivative.

Definition 2.7 (Numerical Derivative) *Let $f\colon \mathbb{R} \to \mathbb{R}$ be a continuous function and Δx a (small) step width. Then, possible numerical derivatives f' at x_0 include:*

- forward derivative*:*
$$f_f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} \tag{2.24}$$
- backward derivative*:*
$$f_b'(x_0) \approx \frac{f(x_0 - \Delta x) - f(x_0)}{-\Delta x} \tag{2.25}$$
- central derivative*:*
$$f_c'(x_0) \approx \frac{f_f'(x_0) + f_b'(x_0)}{2} = \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2 \cdot \Delta x} \tag{2.26}$$

The error induced by these approximations, depending on Δx, is called *truncation error*. While estimations—which also consider higher-order derivatives—show that the central derivative provides the highest accuracy, this thesis focuses on the forward derivative for its simplicity. This is manifested in Definition 2.8, specifying the notion of the derivative of a time series.

Definition 2.8 (Derivative of a Time Series) *Let Ψ be a time series with sampling rate Δx. Further, let $f\colon \mathbb{R} \to \mathbb{R}$ be a function that models Ψ, i.e.,*

$$\forall p \in \mathcal{P}(\Psi)\colon f(\texttt{dpt}(p)) :\!= \texttt{dpv}(p) \tag{2.27}$$

with linear interpolation between measurements (data points) to make it continuous. Then, the derivative of Ψ is the forward derivative of f according to Definition 2.7.

Note that, according to this definition, the derivative of a time series is not defined for its last data point because there is no successor to build a difference with. This restriction might seem grave in examples with very small time series. However, in practice, time series consist of at least several thousand data points, which makes potentially missing out on the last one a negligible drawback.

2.6.2 Numerical Integration

The contents of this subsection are based on [54, pp. 48–58] and [55, pp. 545–549]. They provide more detailed explanations and also error estimations.

A very common interpretation of the *definite integral* of a function $f(x)$ from a to b, i.e.,

$$\int_a^b f(x)dx \tag{2.28}$$

is that it represents the area of the shape made up of the primary axis and f's graph.

Again, since this thesis works with discrete time series, we need to employ a numerical approach. Numerical integration methods usually come down to compartmentalizing $[a, b]$ into n sub-intervals and approximating the integral as the total area of shapes constituted by those sub-intervals. The *midpoint rule*, for instance, sums up the area of rectangles whose heights are given by the value of f at the midpoint of the respective sub-interval. The *trapezoid rule* adds up areas of trapezoids whose parallel sides are given by the values of f at the respective sub-interval's bounds and whose heights are equal to its width. Lastly, *Simpson's rule* approximates f on each sub-interval as a parabola going through the values of f at the sub-interval's bounds and its mid-point. Since the integral of a parabola is easily calculated, the numerical integral of f is then the total sum of definite integrals of approximated parabolas on the sub-intervals.

Error estimations show that, in general, Simpson's rule is the most accurate one. However, due to the fact that time series are not a continuous function, the trapezoid rule comes as a more natural way to represent the numerical integral. Apart from that, it also allows for a more compact formulation—see Definition 2.9.

Definition 2.9 (Numerical Integral Using the Trapezoid Rule) *Let $f : \mathbb{R} \to \mathbb{R}$ be a continuous function, $[a, b]$ an interval and $n \in \mathbb{N}_{\geq 1}$ the number of adjacent sub-intervals which compartmentalize $[a, b]$, but do not necessarily have equal widths. Then, every sub-interval $[a_i, b_i]$ with $1 \leq i \leq n$ gives rise to a trapezoid with area*

$$A_i := \frac{(f(a_i) + f(b_i)) \cdot (b_i - a_i)}{2} \tag{2.29}$$

Therefore, the integral of f from a to b is approximated as

$$\begin{aligned}\int_a^b f(x)dx &\approx \sum_{i=1}^{n} A_i \\ &= \frac{1}{2} \cdot \sum_{i=1}^{n} ((f(a_i) + f(b_i)) \cdot (b_i - a_i))\end{aligned} \tag{2.30}$$

If the sub-intervals have equal widths, the trapezoids' height $(b_i - a_i)$ reduces to a *constant factor $\frac{b-a}{n}$. In that case, numerical integral has a simpler form:*

$$\begin{aligned}\int_a^b f(x)dx &\approx \sum_{i=1}^{n} A_i \\ &= \frac{1}{2}\cdot\sum_{i=1}^{n}\Big((f(a_i)+f(b_i))\cdot\frac{b-a}{n}\Big) \\ &= \frac{b-a}{2n}\cdot\sum_{i=1}^{n}(f(a_i)+f(b_i))\end{aligned} \tag{2.31}$$

Building on that, we are now able to define the (discrete) definite integral of a time series in Definition 2.10.

Definition 2.10 (Integral of a Time Series) *Let Ψ be a time series and $\mathtt{t_a}, \mathtt{t_b} \in \mathbb{Z}$ two points in time. Further, let $f\colon \mathbb{R} \to \mathbb{R}$ be a function that models Ψ, i.e.,*

$$\forall p \in \mathcal{P}(\Psi)\colon f(\mathtt{dpt}(p)) \coloneqq \mathtt{dpv}(p) \tag{2.32}$$

with linear interpolation between measurements (data points) to make it continuous. Then, the integral of Ψ from $\mathtt{t_a}$ to $\mathtt{t_b}$ is the numerical integral of f from $\mathtt{t_a}$ to $\mathtt{t_b}$ according to Definition 2.9.

2.6.3 Linear Regression

The formulas and definitions presented in this subsection are taken from [56, pp. 255–261]. Refer to that reference for a more detailed derivation and a proof of correctness.

In general, the idea behind linear regression is to fit the graph of a function

$$y_r(x) \coloneqq \beta_0 \cdot \varphi_0(x) + \beta_1 \cdot \varphi_1(x) + \cdots + \beta_m \cdot \varphi_m(x) \tag{2.33}$$

to a set of two-dimensional data $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. The *shape functions* $\varphi_i(x)$ with $0 \leq i \leq m$ are assumed as given and dictate the model of regression. The problem is determining the coefficients β_i with $0 \leq i \leq m$ based on the data. The goal is to minimize the sum of squares of errors, i.e., the expression

$$\sum_{i=1}^{n}\big(y_i - y_r(x_i)\big)^2 \tag{2.34}$$

should be as small as possible. This approach is also called the *method of least squares*.

The regression is *linear* because y_r depends linearly on the (unknown) coefficients β_i. One of the most commonly referred to regression problems—which is also employed in this thesis—is *simple* or *univariate linear regression* where y describes a linear model. In other words, we have $\varphi_0(x) \coloneqq 1$, $\varphi_1(x) \coloneqq x$ and $\varphi_k(x) \coloneqq 0$ for $k > 1$, which yields:

$$y_r(x) \coloneqq \beta_0 + \beta_1 \cdot x \tag{2.35}$$

The solution to the minimization problem stated in Equation (2.34), for the regression function from Equation (2.35), is summarized in Definition 2.11. The derivation of this general representation of β_0 and β_1, depending on a concrete data set, goes beyond the scope of this thesis. To get more information on that, refer to the literature mentioned in the beginning of this subsection.

Definition 2.11 (Simple Linear Regression) *Let $\Theta = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ be a data set where at least two x-values are different. Then, the regression line*

$$y_r(x) \coloneqq \beta_0 + \beta_1 \cdot x \tag{2.36}$$

through this data set Θ has unique parameters β_0 and β_1 which minimize the constraint from Equation (2.34) *and are given by*

$$\begin{aligned} \beta_0 &\coloneqq \left(\frac{1}{n}\sum_{i=1}^{n} y_i\right) - \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) \cdot \beta_1 \\ \beta_1 &\coloneqq \frac{\sum_{i=1}^{n}(x_i \cdot y_i) - \left(\frac{1}{n} \cdot \sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} y_i\right)}{\left(\sum_{i=1}^{n} {x_i}^2\right) - \frac{1}{n} \cdot \left(\sum_{i=1}^{n} x_i\right)^2} \end{aligned} \tag{2.37}$$

In order to characterize the regression line during an interval ${}_i\pi_j$ of a time series Ψ using this definition, we first need to specify how to construct a data set $\Theta = (\{x, y\}) \subset \mathbb{R} \times \mathbb{R}$ from ${}_i\pi_j$. Evidently, the value components of data points from ${}_i\pi_j$ can be assigned directly to the y component. For the x component, we map the time components to the duration between the start of the interval ${}_i\pi_j$ and the point in time they represent. It is crucial, however, to observe that the slope of the resulting regression line heavily depends on the time unit in which this time difference is determined.

The slope of the regression line shrinks proportionally to the resolution of the unit in which the time difference is expressed. For example, consider two data points which are 15 minutes apart—$\{(\texttt{10:00}, 0), (\texttt{10:15}, 15)\}$. If the time difference is measured in minutes—i.e., with $\Theta = \{(0, 0), (15, 15)\}$—, they define a regression line of $y_r(x) = 1 \cdot x$. If, on the other hand, the time difference is measured in seconds (which provide a temporal resolution that is 60-times higher than with minutes), we have $\Theta = \{(0, 0), (900, 15)\}$ and $y_r(x) = \frac{1}{60} \cdot 1 \cdot x = 0.01\dot{6} \cdot x$.

Therefore, when calculating the regression line over a time interval, it is necessary to determine the unit in which time differences are calculated based on the concrete time series. A relatively simple way to do this is presented by Algorithm 2.1. It approximates the temporal resolution of a time series—or, depending on the input, an interval—by examining the average time difference between pairwise adjacent data points (line 2). In increasing order of resolution, it gauges whether the current time unit supported by DTSQL describes this time difference best (line 3). It considers this to be the case if the difference, expressed in the respective unit, is at least 0.85 (lines 4–5). The algorithm rounds up to 1 because, for instance, for an average time difference (sampling rate) of six days (≈ 0.8571 weeks), computing a regression line based on `weeks` yields more natural results than when working with `days`. If no supported time unit exhibits a sufficiently high resolution, the algorithm defaults to `millis` (line 8).

Algorithm 2.1: Inference of Temporal Resolution From a Time Series

Input: time series or set of data points $\Psi = \langle p_1, \ldots, p_n \rangle$
Output: inferred unit of sampling rate
$u \in \{\texttt{weeks}, \texttt{days}, \texttt{hours}, \texttt{minutes}, \texttt{seconds}, \texttt{millis}\}$

```
1 function resolution(Ψ):
2     $t_\Delta \leftarrow \underset{p_i,\, p_{i+1} \in \Psi}{\text{avg}} (\texttt{dpt}(p_{i+1}) - \texttt{dpt}(p_i))$
3     foreach $u \in \langle \texttt{weeks}, \texttt{days}, \texttt{hours}, \texttt{minutes}, \texttt{seconds}, \texttt{millis} \rangle$ do
4         if $\texttt{duration}(t_\Delta,\, u) \geq 0.85$ then
5             return $u$
6         end
7     end
8     return millis
9 end
```

Finally, Definition 2.12 formalizes the just now established notion of regression lines over intervals of time series under utilization of Algorithm 2.1.

Definition 2.12 (Linear Regression on Time Series) *Let Ψ be a time series and ${}_i\pi_j$ an interval over Ψ. Let Θ further be a data set that contains the values of data points in ${}_i\pi_j$, associated with the duration between* $\mathsf{t_i}$ *and their time component, determined in the unit returned by Algorithm 2.1. It is defined as follows:*

$$\begin{aligned}\Theta \coloneqq \{(\Delta x, y) \mid\ & p \in \texttt{ps}_\Psi({}_i\pi_j) \\ & \wedge\ u = \texttt{resolution}(\texttt{ps}_\Psi({}_\texttt{i}\pi_\texttt{j})) \\ & \wedge\ \Delta x = \texttt{duration}(\texttt{dpt}(p) - \texttt{intstart}({}_i\pi_j),\, u) \\ & \wedge\ y = \texttt{dpv}(p)\}\end{aligned} \tag{2.38}$$

Then, the regression line of Ψ with respect to the interval ${}_i\pi_j$ is defined as the regression line of the data set Θ as per Definition 2.11.

CHAPTER 3

Collection of Requirements

This chapter elaborates on the requirements that DTSQL, the query language developed as part of this thesis, aims to fulfill. The requirements were collected mainly in the form of practice-oriented use cases, in collaboration with the domain experts introduced in Section 1.3 (Solution Concept). The identified use cases will be explained and illustrated. Along with descriptions and figures, each dedicated subsection also contains an example of what a query covering a respective use case is expected to return, given a concrete input time series.

3.1 Non-Functional Requirements

From a project management perspective, an iterative approach was chosen to capture requirements and monitor their implementation—rather than producing an exhaustive functional specification document at the start. This greatly facilitates ensuring that developments do not diverge from the domain experts' expectations. In practice, we held regular meetings to discuss design as well as implementation aspects of the language developed over the course of this thesis.

A central demand is that the system's core functionality—i.e., the query mechanisms—should not rely on any other database systems, it should be independent. The reason for this is that query interfaces of databases may be volatile and continuously maintaining the system to support the latest changes is resource-intensive. Additionally, materializing the DTSQL implementation as a translation to an existing query language would mean restricting ourselves to the constructs and features provided by this language. Apart from that, the underlying primary storage mechanism employed is subject to change—making the query language depend on it would associate switching to another database system with significant effort.

Therefore, not only should the system not rely on query routines of other temporal databases, but it should also be able to process time series data from heterogeneous sources. In other words, a generic, extensible architecture should make it possible to implement data adapters which enable executing queries over arbitrary data sources.

A syntactical requirement is to be as intuitive as possible. This is, of course, a highly subjective constraint. In order to match the domain experts' perception of intuitiveness, new language features are reviewed as part of the regular meetings.

Finally, the system should be developed with performance in mind, but that is not a primary concern. The highest priority is bestowed on being able to use DTSQL to express concepts that are relevant to the domain. Certainly, all features should be implemented as efficiently as possible, reducing runtime (and space) complexity by exploiting the structure of given problems. However, conducting extensive performance profiling and optimization of every computational step is not required at this point.

3.2 Functional Requirements

This section provides a detailed description of the identified use cases which should be expressible in DTSQL.

3.2.1 UC1: Global Aggregates

The ability to compute aggregates with respect to the value dimension—e.g., in order to summarize the power consumption of a production machine—is quintessential. This concerns mainly the statistical measures arithmetic mean, standard deviation, minimum, maximum, sum, and count. In the case of global aggregates, the entire time series is considered for the calculation.

Example

In Figure 3.1, a sample time series is depicted which serves as input for the computation of the above-mentioned global aggregates.

Expected Result

The global aggregate values to be computed from the time series visualized in Figure 3.1—i.e., the expected result values of a corresponding DTSQL query—are shown in Table 3.1.

3.2.2 UC2: Local Aggregates

This use case is almost identical to global aggregates. The only difference is that it should be possible to specify a start and/or end date of the range to consider when calculating the aggregate values (instead of the entire time series).

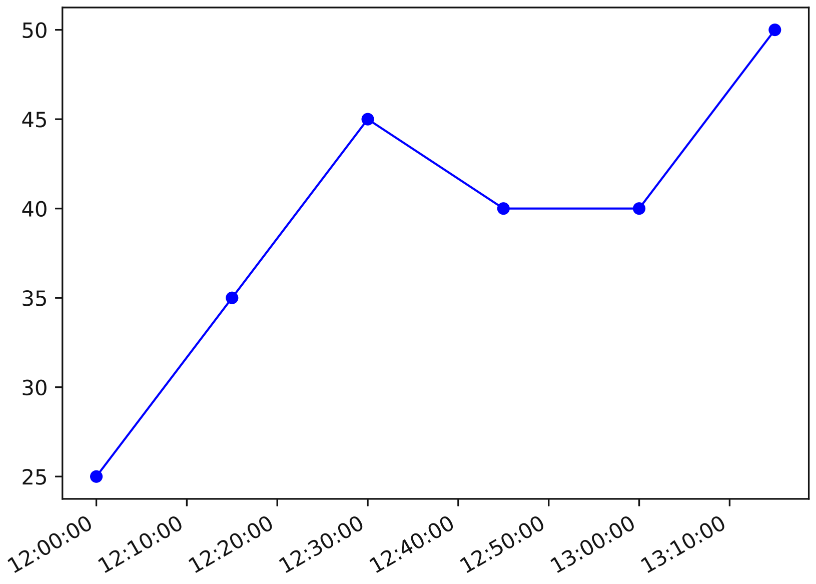


Figure 3.1: Illustration of DTSQL Use Case “UC1: Global Aggregates”

aggregate	value
average	39.1667
standard deviation	7.8617
minimum	25
maximum	50
total (sum)	235
count	6

Table 3.1: Result Aggregates of “UC1: Global Aggregates” Given Figure 3.1

Example

In Figure 3.2, a sample time series is depicted which serves as input for the the computation of the above-mentioned local aggregates for data points between `12:10:00` and `13:10:00`.

Expected Result

The local aggregate values for data points between `12:10:00` and `13:10:00` to be computed from the time series visualized in Figure 3.2—i.e., the expected result values of a corresponding DTSQL query—are shown in Table 3.2.

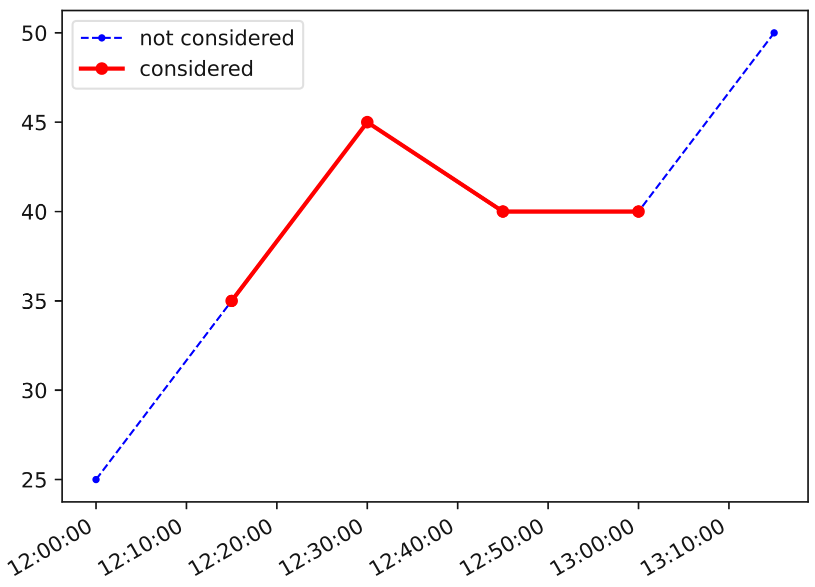


Figure 3.2: Illustration of DTSQL Use Case "UC2: Local Aggregates"

aggregate	value
average	40
standard deviation	3.5356
minimum	35
maximum	45
total (sum)	160
count	4

Table 3.2: Result Aggregates of "UC2: Local Aggregates" Given Figure 3.2

3.2.3 UC3: Temporal Aggregates

Assume a query or manual analysis yields multiple intervals capturing events of note—e.g., passive periods of a machine. In order to gain insights such as the average or total amount of time spent in a passive state, it should be possible to compute the aggregates mentioned in "UC1: Global Aggregates" and "UC2: Local Aggregates", but with respect to the duration of the previously identified intervals (instead of the values of the data points contained in the intervals).

Example

Suppose there have been identified three, in any way relevant, intervals. They are highlighted by red, green, and magenta markers in Figure 3.3. Calculate all supported types of aggregate values from "UC1: Global Aggregates" and "UC2: Local Aggregates", but with respect to the intervals' durations.

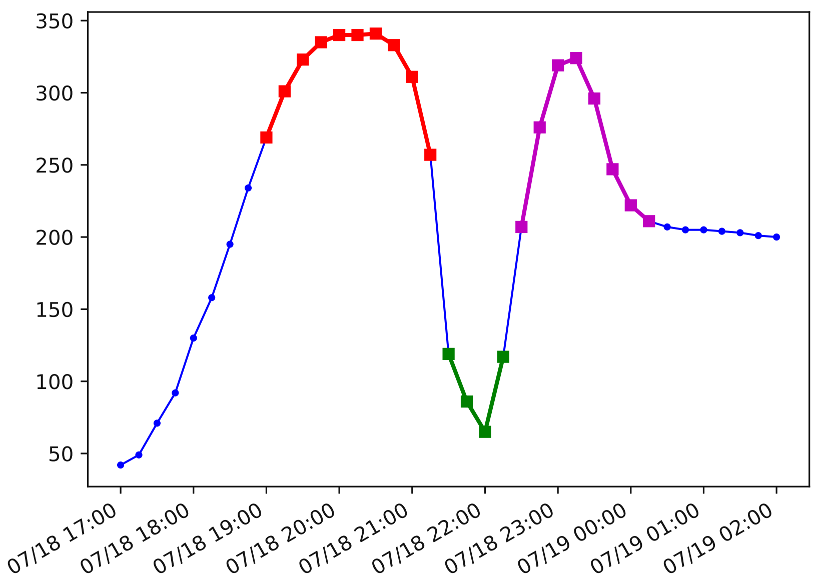


Figure 3.3: Illustration of DTSQL Use Case "UC3: Temporal Aggregates"

Expected Result

Table 3.3 gives a clearer definition of the three intervals of interest highlighted by Figure 3.3. This table simplifies the calculation of the desired aggregate values is. They are depicted in Table 3.4.

interval	**start**	**end**	**duration**
red	`07/18 19:00`	`07/18 21:15`	135
green	`07/18 21:30`	`07/18 22:15`	45
magenta	`07/18 22:30`	`07/19 00:15`	105

Table 3.3: Intervals Processed By "UC3: Temporal Aggregates" Given Figure 3.3

3.2.4 UC4: Numerical Integral

A recurring task is observing the power measurements captured by a sensor and inferring the energy (the work) that occurred during the process. Since power is the time derivative of work, it is possible to approximate the energy consumption by calculating the numerical definite integral of the power measured over time.

aggregate	value
average	95
standard deviation	37.4166
minimum	45
maximum	135
total (sum)	285
count	3

Table 3.4: Result Aggregates of "UC3: Temporal Aggregates" Given Table 3.3

Example

Approximate the energy consumption during the period from `12:45` until `13:45`—visualized by Figure 3.4, measurements in watts (W)—in kilojoule (kJ) by computing during this period.

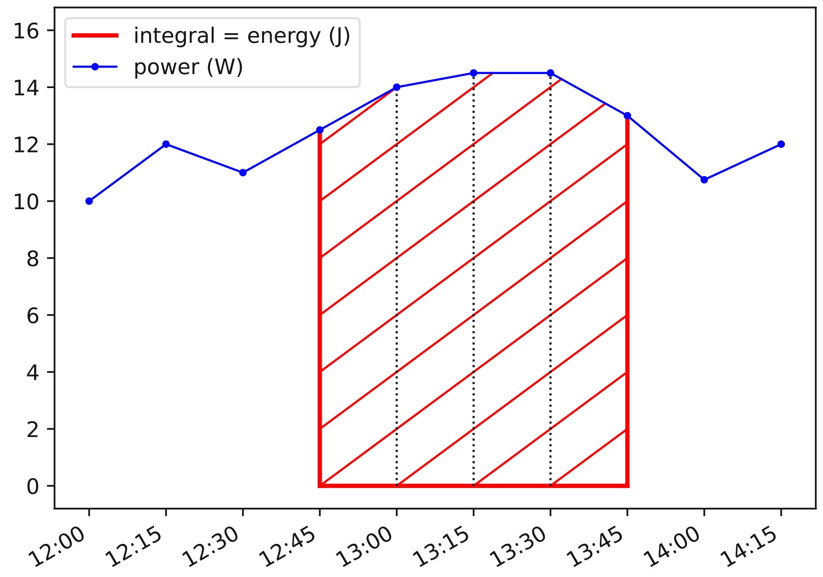


Figure 3.4: Illustration of DTSQL Use Case "UC4: Numerical Integral"

Expected Result

Table 3.5 provides an in-depth view of the period in question. As Figure 3.4 shows, there is a sampling rate of 15 minutes, i.e., 900 seconds. Hence, the integral can be calculated by applying the trapezoidal rule as defined in Section 2.6.2 (Numerical Integration). This calculation is depicted in Table 3.5. From that, it is evident that the integral amounts to 50175 J, or 50.175 kJ.

time	power (W)	cumulative integral (J)
`12:45`	12.5	0
`13:00`	14.0	$0 + ((12.5 + 14.0) \cdot 900 \cdot 0.5) = 11925$
`13:15`	14.5	$11925 + ((14.0 + 14.5) \cdot 900 \cdot 0.5) = 11925 + 12825 = 24750$
`13:30`	14.5	$24750 + ((14.5 + 14.5) \cdot 900 \cdot 0.5) = 24750 + 13050 = 37800$
`13:45`	13.0	$37800 + ((14.5 + 13.0) \cdot 900 \cdot 0.5) = 37800 + 12375 = \mathbf{50175}$

Table 3.5: Interval Processed By "UC4: Numerical Integral" Given Figure 3.4

3.2.5 UC5: Threshold Filters

A very common, indispensable, filter operation is based on a threshold value. This makes it possible to exclude outliers or focus in on a specific range of values. Hence, there should be a way to filter out data points whose values are (not) greater than or (not) less than a fixed threshold or aggregate value. It should also be possible to apply multiple such constraints at once.

Example

Find all sensor data points of the time series depicted by Figure 3.5 whose values are greater than 4, but not greater than 10.

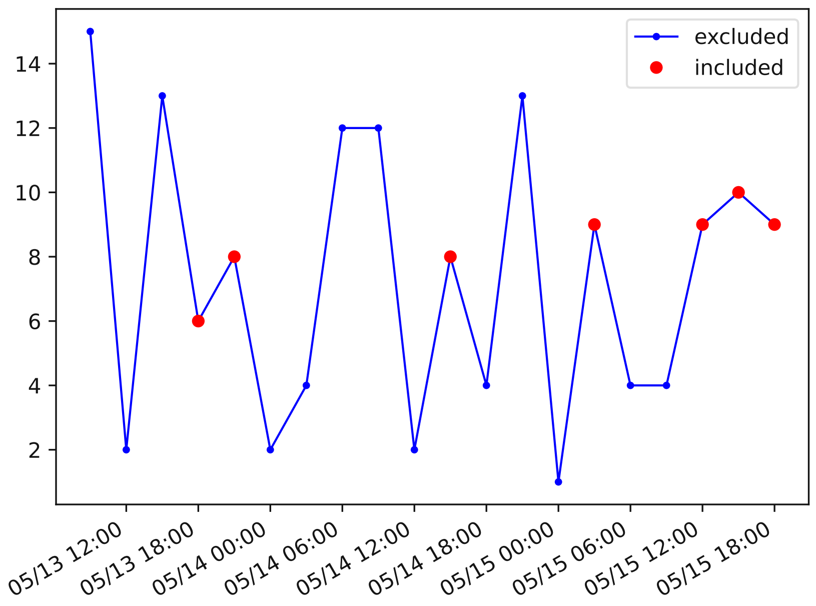


Figure 3.5: Illustration of DTSQL Use Case "UC5: Threshold Filters"

Expected Result

The example specification asks for a filter that only includes data points whose values are between 4.0 (exclusively) and 10 (inclusively). Table 3.6 depicts all data points which satisfy these criteria, as already indicated by the color red in Figure 3.5.

time	value
`05/13 18:00`	6
`05/13 21:00`	8
`05/14 15:00`	8
`05/15 03:00`	9
`05/15 12:00`	9
`05/15 15:00`	10
`05/15 18:00`	9

Table 3.6: Result Data Points of "UC5: Threshold Filters" Given Figure 3.5

3.2.6 UC6: Temporal Filters

This type of filter is the equivalent of "UC5: Threshold Filters" in the temporal dimension. By filtering out data points which were recorded (not) before or (not) after specific points in time, one is able to focus in on relevant periods.

Example

Filter out all data points from the time series illustrated by Figure 3.6 which were *not* recorded between `2022-05-14 10:45` and `2022-05-15 00:00`.

Expected Result

According to the example specification, all data points that were recorderd before `2022-05-14 10:45` or after `2022-05-15 00:00` should be discarded. Table 3.7 depicts those data points which satisfy these criteria, as already indicated by the color red in Figure 3.6.

time	value
`2022-05-14 12:00`	2
`2022-05-14 15:00`	8
`2022-05-14 18:00`	4
`2022-05-14 21:00`	13
`2022-05-15 00:00`	1

Table 3.7: Result Data Points of "UC6: Temporal Filters" Given Figure 3.6

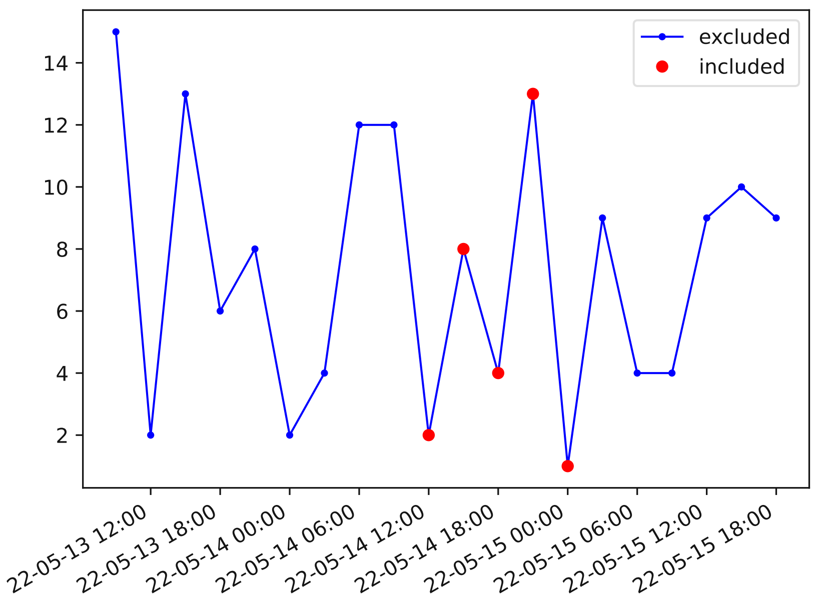


Figure 3.6: Illustration of DTSQL Use Case "UC6: Temporal Filters"

3.2.7 UC7: Threshold Events

Events are generally concerned with specifying intervals for which some condition(s) should hold. Similar to "UC5: Threshold Filters", for this use case it is relevant to detect periods in which data point values are continuously (not) above or (not) below some threshold. Based on intervals which were detected that way, further calculation can be made—e.g., on periods with energy levels that are higher than the average.

A general note on this and all subsequent use cases which yield intervals: DTSQL should not only be able to return all detected intervals, but also the minimum and maximum ones (with respect to length).

Example

Find the maximum interval in the time series depicted by Figure 3.7 during which the recorded value was consistently above 1500.

Expected Result

As already illustrated by Figure 3.7, there are three maximal intervals with values consistently higher than 1500. They are denoted by red, magenta, and green square markers. Table 3.8 gives a more detailed view on these three intervals. It makes evident that the result to the question posed the red interval from `01:15` until `03:30` with a length of 135 minutes.

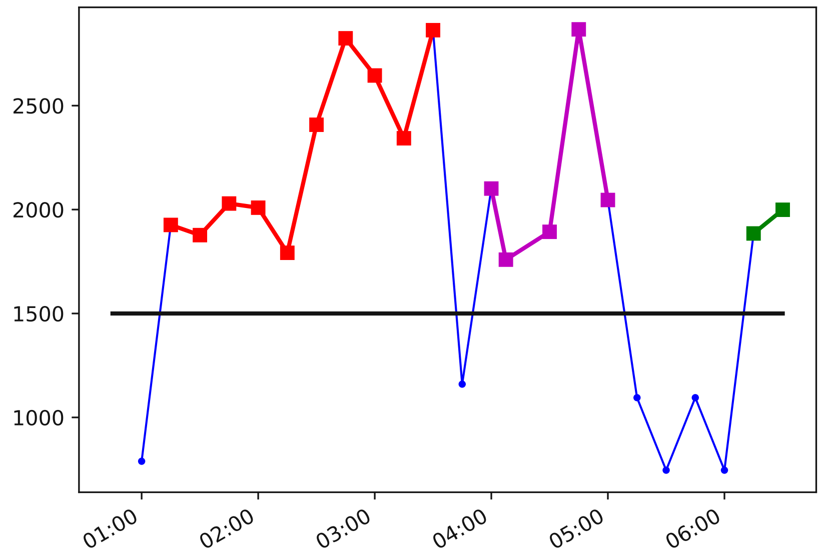


Figure 3.7: Illustration of DTSQL Use Case “UC7: Threshold Events”

interval	**start (HH:MM)**	**end (HH:MM)**	**length (minutes)**
red	`01:15`	`03:30`	135
magenta	`04:00`	`05:00`	60
green	`06:15`	`06:30`	15

Table 3.8: Result Interval(s) of “UC7: Threshold Events” Given Figure 3.7

3.2.8 UC8: Deviation Events

An operation related to “UC7: Threshold Events” is the detection of periods where values are in a specifiable range of an arbitrary reference value. This range of maximal deviation from the reference value, depending on the purpose of the query, may be specified in absolute or relative (percentage) numbers.

Example

Find all intervals in the time series depicted by Figure 3.8 with a recorded power that is consistently within a range of ±10 % of the global average power.

Expected Result

The global average power recorded in the time series visualized by Figure 3.8 is approximately 165.63. The resulting valid interval represented by 165.63 ± 10 % has a lower bound of roughly 149.07 and an upper bound of about 182.19. There are two intervals which satisfy this restriction. They have been annotated using red and green square

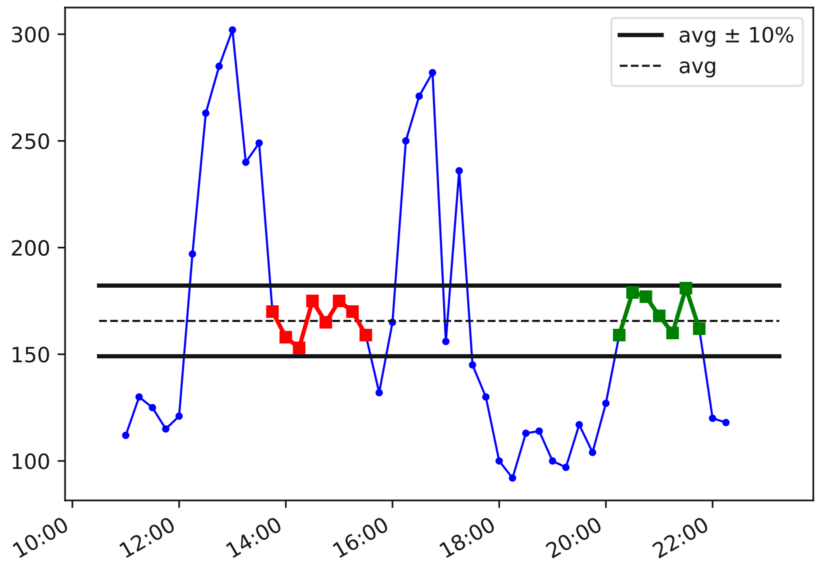


Figure 3.8: Illustration of DTSQL Use Case "UC8: Deviation Events"

markers. A clearer justification as to why the red interval from `13:45` until `15:30` and the green interval from `20:15` until `21:45` are valid solutions to the problem can be observed in Table 3.9. Their respective minimum and maximum values are within the interval $[149.07, 182.19]$, hence they must satisfy the condition.

interval	start	end	minimum	maximum
red	`13:45`	`15:30`	153	175
green	`20:15`	`21:45`	159	181

Table 3.9: Result Intervals of "UC8: Deviation Events" Given Figure 3.8

3.2.9 UC9: Constant Events

It is highly valuable for time series analysts to be able to capture intervals with approximately constant values. Manually, they do so mostly by examining the magnitude of fluctuations displayed by the data as well as their general trend. Since the notion of what is considered constant varies from subject to subject, the query language should also provide means of specifying tolerance levels.

Example

The task is to find the longest constant interval of the time series shown in Figure 3.9.

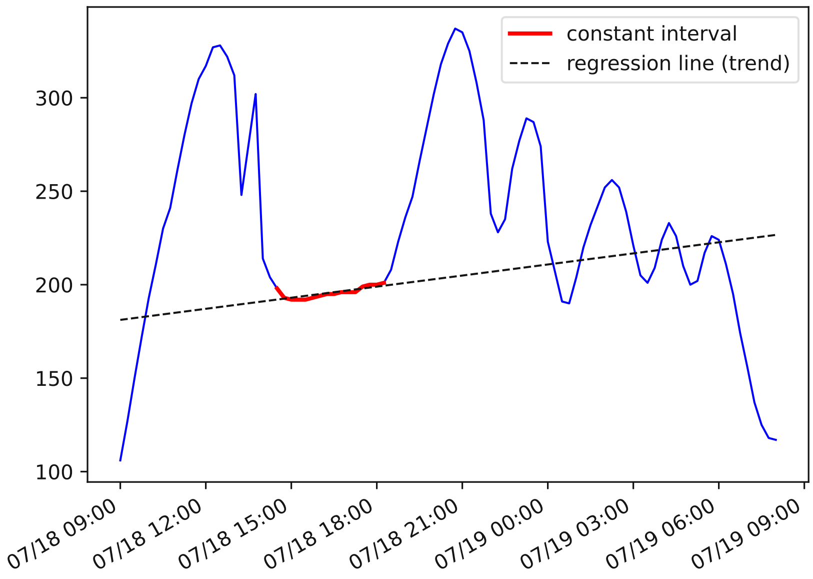


Figure 3.9: Illustration of DTSQL Use Case "UC9: Constant Events"

Expected Result

As already indicated by the red period in Figure 3.9, the longest constant interval is between `07/18 14:30` and `07/18 18:15`. The concrete data points it contains are given in Table 3.10. The figure also features the regression line for those data points, as defined in Section 2.6.3 (Linear Regression). While its trend is generally increasing, it should still be deemed constant due to the small relative deviations shown in Table 3.10.

3.2.10 UC10: Monotonic Events

Similar to "UC9: Constant Events", detecting roughly monotonically increasing or decreasing intervals is particularly important, too. For instance, they may characterize the switching on of a machine, or an unusual spike in energy expenditure. Not every increase is worth investigating, however. Therefore, DTSQL should provide ways of expressing that an increase must exhibit a minimum magnitude.

Example

Given the time series depicted by Figure 3.10, find all intervals that represent a monotonic increase of at least 800 %.

Expected Result

Figure 3.10 already suggests that their are three intervals which may describe a monotonic increase satisfying the given conditions. They may be examined closer in Table 3.11. It

time	value
07/18 14:30	198
07/18 14:45	193
07/18 15:00	192
07/18 15:15	192
07/18 15:30	192
07/18 15:45	193
07/18 16:00	194
07/18 16:15	195
07/18 16:30	195
07/18 16:45	196
07/18 17:00	196
07/18 17:15	196
07/18 17:30	199
07/18 17:45	200
07/18 18:00	200
07/18 18:15	201

Table 3.10: Result Interval of "UC9: Constant Events" Given Figure 3.9

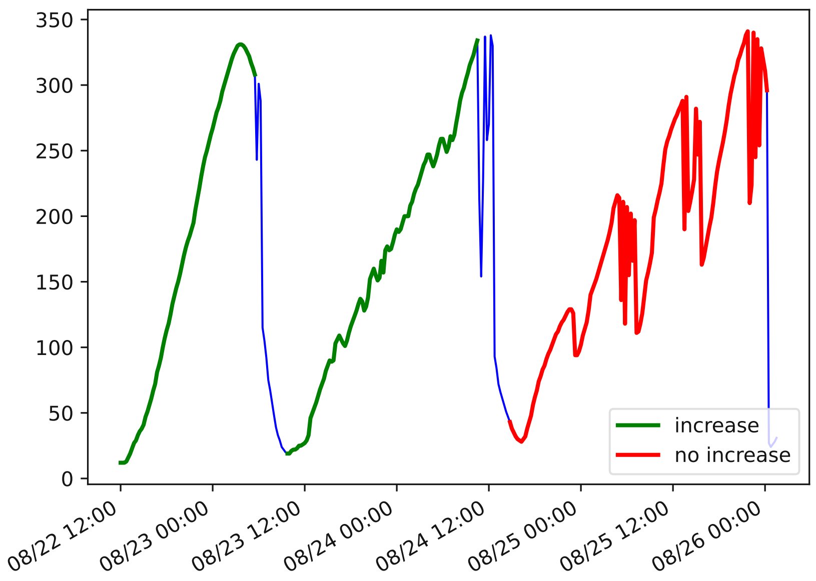


Figure 3.10: Illustration of DTSQL Use Case "UC10: Monotonic Events"

is evident that the first green interval represents a monotonic increase—even a strictly monotonic one. The second green interval does exhibit temporary decreases, but they are miniscule enough to be overlooked. The last one, the red interval as a whole, however, is different. While it displays an increase of more than 800 %, it is not a (roughly) monotonic one since there are too many too pronounced decreases in between. Note that there are sub-intervals of the red interval which individually might be considered monotonic increases using different parameters.

start	end	start value	end value	increase
`08/22 12:00`	`08/23 05:30`	12	308	2467 %
`08/23 09:45`	`08/24 10:30`	19	334	1658 %
`08/24 10:30`	`08/26 00:15`	43	296	887 %

Table 3.11: (Potential) Result Intervals of "UC10: Monotonic Events" Given Figure 3.10

3.2.11 UC11: Duration Constraints for Events

For any event supported by DTSQL (threshold, deviation, constant, increase, decrease), one should be able to impose duration constraints on their definition. This proves useful since some phenomena are only interesting if they occur for or within a certain amount of time. For instance, a significant increase in pressure might be much more noteworthy if it occurs within seconds rather than, as expected, slowly over minutes.

Example

For this example, we slightly modify the one from "UC7: Threshold Events". We are now looking for all intervals in the time series depicted by Figure 3.11 during which the recorded value was consistently above 1500, but for a *maximum* period of 60 minutes. Since the underlying time series is the same as with "UC7: Threshold Events", the plot is almost identical. For clarity, the three intervals have been annotated according to their satisfaction of the duration constraint.

Expected Result

As already illustrated by Figure 3.11, there are three maximal intervals with values consistently higher than 1500. They are, again, denoted using red, magenta, and green color. The figure also highlights that only two intervals, the magenta and green one, satisfy the constraint of exhibiting a length of not more than 60 minutes. Table 3.8 gives more insight into why this is the case. It shows that both the magenta interval—from `04:00` until `05:00`—with a length of 60 minutes as well as the green interval—from `06:15` until `06:30`—with a duration of 15 minutes constitute the expected result. The red one—from `01:15` until `03:30`—with a length of 135 minutes violates the constraint.

3.2.12 UC12: Binary Event Sequences

One of the central aspects making DTSQL useful is the ability to express temporal sequences between periods which were detected as a result of event definitions. For

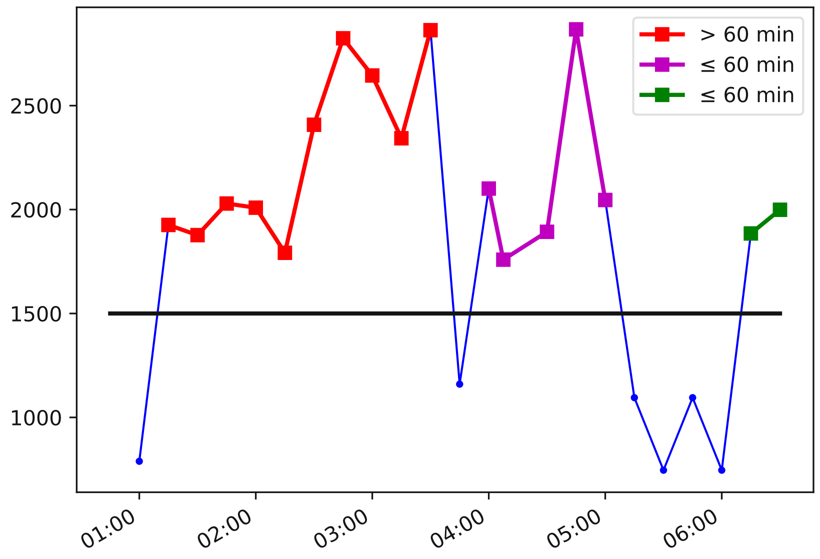


Figure 3.11: Illustration of DTSQL Use Case "UC11: Duration Constraints for Events"

interval	**start (HH:MM)**	**end (HH:MM)**	**length (minutes)**
red	`01:15`	`03:30`	135
magenta	`04:00`	`05:00`	60
green	`06:15`	`06:30`	15

Table 3.12: Result Intervals of "UC11: Duration Constraints for Events" Given Figure 3.11

instance, this might allow the specification of a composite interval which captures a normal state of operation after turning on a machine—i.e., a constant period *following* a monotonic increase. Analogously, sequences expressing that an event occurs *before* another one should also be supported. Since, in this use case, there are always exactly two events involved, they make up *binary* event sequences.

Example

In this example, two event definitions are present. One is concerned with detecting periods where all values are greater than or equal to 220. The other one describes constant periods. Now, given the time series illustrated by Figure 3.12, find composite intervals where a period of the first event directly follows an interval from the second event.

Expected Result

As it can be observed in Figure 3.12, there are multiple intervals where the two conditions from the example specification are satisfied individually. They are denoted using green

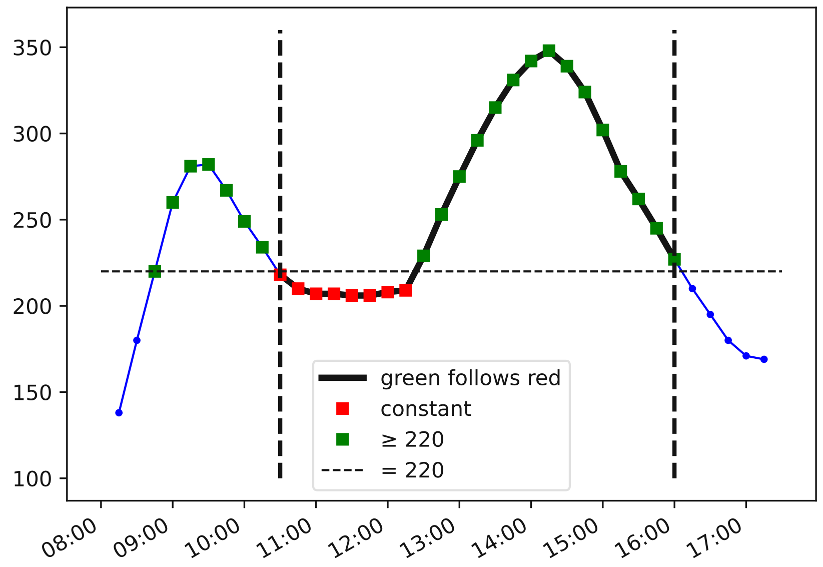


Figure 3.12: Illustration of DTSQL Use Case "UC12: Binary Event Sequences"

and red square markers, respectively. The interval which is actually requested in this example, however, is a combination of those where a green interval appears directly after a red one. In this instance, this happens exactly once. The exact boundaries of the intervals involved are depicted in Table 3.13. It also shows that the expected result interval shares its lower bound `10:30` with the red interval and its upper bound `16:00` with the green interval.

interval	**start**	**end**
red	`10:30`	`12:15`
green	`12:30`	`16:00`
green `follows` red	`10:30`	`16:00`

Table 3.13: Result Interval(s) of "UC12: Binary Event Sequences" Given Figure 3.12

3.2.13 UC13: Time Tolerance for Event Sequences

Expanding on "UC12: Binary Event Sequences" and similar to "UC11: Duration Constraints for Events", some instances require the ability to allow for a certain amount of time to transpire between two intervals constituting a binary event sequence. This is necessary because realistic data only seldom has clear-cut transitions between individual phenomena. Furthermore, some concepts are defined using these kinds of *time-gap constraints*, as the example below shows.

Example

Detect an *active power trip*, which happens when the active power is above 1.5 megawatt for a period of at least ten seconds, after it has been below 0.15 megawatt for at least one minute. The time-gap between two such periods must not exceed three seconds. This could also be referred to as a *power surge*. An examplary time series visualizing an active power trip is shown in Figure 3.13.

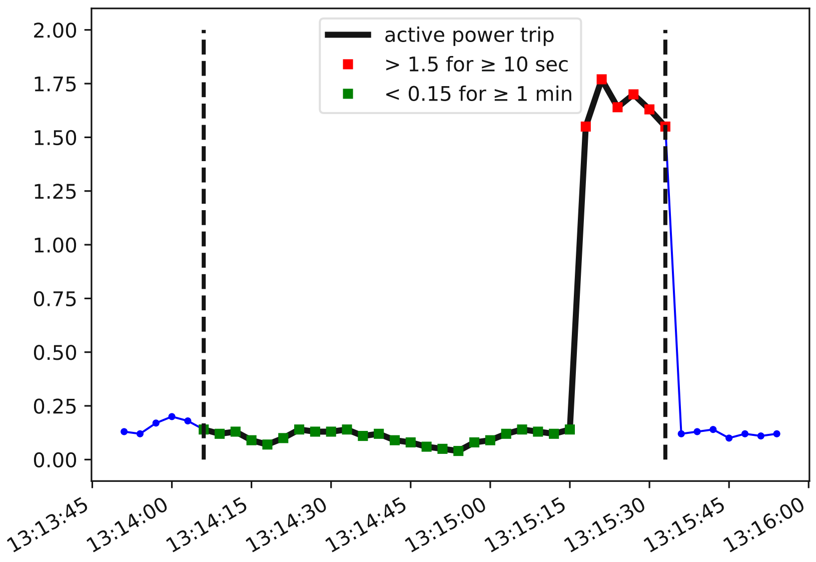


Figure 3.13: Illustration of DTSQL Use Case "UC13: Time Tolerance for Event Sequences"

Expected Result

The intervals constituting the active power trip are visualized in Figure 3.13 and broken down in Table 3.14. The green interval—from `13:14:06` until `13:15:15`—satisfies the condition of the example specification with its duration of 69 seconds, which is more than one minute. The red interval satisfies the other condition because its duration of 15 seconds—from `13:15:18` until `13:15:33`—is longer than 10 seconds. Furthermore, the time-gap between the green and red interval—from `13:15:15` until `13:15:18`—is exactly three seconds, satisfying the final condition. Therefore, the expected result capturing the active power trip is the interval from `13:14:06` until `13:15:33`.

3.2.14 UC14: N-Ary Event Sequences

This use case is an extension of "UC13: Time Tolerance for Event Sequences". Sometimes it is not sufficient to detect binary sequences of events. For instance, in order to capture the whole operational cycle of a machine (turn on, run, turn off), a ternary event sequence

interval	start	end	duration
green	`13:14:06`	`13:15:15`	69 sec
red	`13:15:18`	`13:15:33`	15 sec
red `follows` green	`13:14:06`	`13:15:33`	87 sec

Table 3.14: Result Interval(s) of "UC13: Time Tolerance for Event Sequences" Given Figure 3.13

is required. Therefore, it is necessary for DTSQL to be able to express such—and, in general, n-ary—sequences of events.

Example

Using the time series displayed in Figure 3.14 as base data, detect a three-stage process where values are at first below or equal to 70, then above 70 and below or equal to 150, and finally above 150.

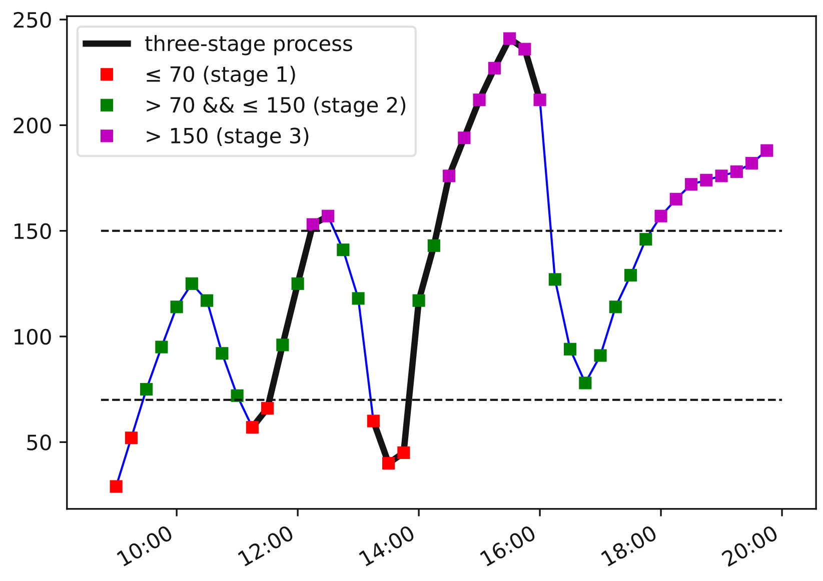


Figure 3.14: Illustration of DTSQL Use Case "UC14: N-Ary Event Sequences"

Expected Result

This example is very similar to the one in "UC12: Binary Event Sequences". In this instance, however, there are two result intervals satisfying the example specification instead of only one. Table 3.15, in conjunction with Figure 3.14, illustrate why this is the case. The three different stages defined by threshold values 70 and 150, occur several times across the time series. As it can be observed in the figure, the required event sequence *stage 1* $\rightarrow$ *stage 2* $\rightarrow$ *stage 3* occurs exactly twice—from `11:15` until `12:30` as well as between `13:15` and `16:00`. Hence, these two intervals represent the expected result of a query corresponding to this use case.

#	interval	start	end
1	stage 1	`11:15`	`11:30`
	stage 2	`11:45`	`12:00`
	stage 3	`12:15`	`12:30`
	stage 3 `follows` stage 2 `follows` stage 1	`11:15`	`12:30`
2	stage 1	`13:15`	`13:45`
	stage 2	`14:00`	`14:15`
	stage 3	`14:30`	`16:00`
	stage 3 `follows` stage 2 `follows` stage 1	`13:15`	`16:00`

Table 3.15: Result Intervals of “UC14: N-Ary Event Sequences” Given Figure 3.14

CHAPTER 4

Query Language Specification

One of the core contributions of this thesis is the design—i.e., formal specification—of a query language for time series data that is expressive enough to cover the use cases outlined in Chapter 3. This chapter provides such a specification of DTSQL by means of three sections. At first, Section 4.1 introduces an abstract syntax of the language. It formalizes the structure and properties of the individual components of a query and provides an intuitive understanding of their meaning. Secondly, a precise definition of the semantics of those language features is presented in Section 4.2. Lastly, Section 4.3 presents a concrete grammar that is compatible with the abstract syntax specified in Section 4.1. This grammar is a plain-text representation of the language which can be used to create parsers for DTSQL queries. This will be leveraged in Chapter 5 (Reference Implementation).

4.1 Language Syntax

This section defines the abstract syntax of DTSQL and queries expressed in that language. At first, the overall structure of DTSQL queries is presented, introducing their various components. The subsequent subsections are each dedicated to one of these component. They provide concrete definitions of their syntax and properties, along with demonstrative examples.

4.1.1 DTSQL Query

A DTSQL query consists of five components:

1. A set of *samples*. Samples represent scalar values expressed by aggregation functions such as an average or sum. These values may be referenced (reused) when defining other query components—e.g., in filters or events—or serve as overall query result.

2. A *filter* specification that excludes irrelevant data points of the input time series from the query evaluation.

3. A set of *events*. Events are a declarative specification of intervals capturing specific phenomena in a time series. For instance, intervals with values below a given threshold, periods with (approximately) constant values or monotonic increases may be described using events.

4. A *selection* consisting of a composition operator that defines temporal relations between detected events. This component allows query creators to express that events have to occur in a certain order of succession to be part of the query result.

5. A *yield* statement which ultimately determines the query result. The actual value of this statement decides whether the result is made up of data points, intervals (their lower and upper bounds) or sample values.

Out of these five components, only the last one—the yield statement—is mandatory. The others are optional in the sense that the set of samples may be empty, the filter specification may include not a single filter, the set of events may be empty and the selection may be blank.

Definition 4.1 depicts a holistic overview of the syntax of a DTSQL query. Keep in mind that the notations used in this definition will be explained and formalized more precisely in the following subsections.

Definition 4.1 (DTSQL Query) *Given an input time series Ψ, a DTSQL query κ over Ψ is defined as the quintuple*

$$\begin{aligned}\kappa &:= (\mathcal{S}, \Phi, \varXi, \Omega, \Upsilon) \\ &= (\underbrace{\{s_1, s_2, \dots\}}_{\mathcal{S}}, \underbrace{(\mathcal{F}, \varphi)}_{\Phi}, \underbrace{\{\varepsilon_1, \varepsilon_2, \dots\}}_{\varXi}, \underbrace{\omega}_{\Omega}, \underbrace{(\gamma, d)}_{\Upsilon})\end{aligned} \tag{4.1}$$

where $\mathcal{S}$ is a set of samples, Φ a filter specification with filters $\mathcal{F}$ connected by the formula φ, $\varXi$ a set of event definitions, Ω a composition operator relating detected intervals to each other and Υ a yield statement of type γ and result definition parameter d (e.g., for sample references).

4.1.2 Samples

A *sample* is a scalar, real value that can be computed given a time series Ψ. The computation of a sample is never reliant upon other, previously computed samples, i.e., they are independent of each other. Essentially, samples are *aggregate* values that can be (re-)used to make other components of the query depend on them—e.g., to compose filters or define events instances (see Section 4.1.3 and Section 4.1.4, respectively). They may also be utilized in the yield component so that a query returns one or more sample values.

DTSQL supports three types of samples, *value aggregates*—which may be *global* or *local*—and *temporal aggregates*. Global value aggregates operate on all values of a time series and are specified in Definition 4.2.

Definition 4.2 (Global Value Aggregates) *Let Ψ be a time series and $s \in \mathbb{R}$ a sample. Then, s is a* global value aggregate *if it was defined using one of the following function symbols:*

1. $\texttt{max}_\Psi$*: greatest value of all data points in Ψ*
2. $\texttt{min}_\Psi$*: lowest value of all data points in Ψ*
3. $\texttt{avg}_\Psi$*: arithmetic mean of all data point values in Ψ*
4. $\texttt{count}_\Psi$*: number of data points in Ψ*
5. $\texttt{sum}_\Psi$*: sum of all data point values in Ψ*
6. $\texttt{integral}_\Psi$*: definite integral of the discrete function modelled by Ψ*
7. $\texttt{stddev}_\Psi$*: population standard deviation of all data point values in Ψ*

Local value aggregates are very similar to global ones. They only differ in the fact that they are computed on a subset of the input time series that is specified by temporal boundary values—see Definition 4.3.

Definition 4.3 (Local Value Aggregates) *Let Ψ be a time series and $\texttt{t}_1, \texttt{t}_2 \in \mathbb{Z}$ with $\texttt{t}_2 \geq \texttt{t}_1$ be two arbitrary points in time. Further, let Ψ' be a time series containing only data points from Ψ whose time components are between $\texttt{t}_1$ and $\texttt{t}_2$, such that*

$$\mathcal{P}(\Psi') = \{p \in \mathcal{P}(\Psi) \mid \texttt{dpt}(p) \geq \texttt{t}_1 \wedge \texttt{dpt}(p) \leq \texttt{t}_2\} \tag{4.2}$$

holds. Then, the sample $s \in \mathbb{R}$ is a local value aggregate *if it was defined using one of the following function symbols:*

1. $\texttt{max}_\Psi(\texttt{t}_1, \texttt{t}_2)$*: greatest value of all data points in Ψ'*
2. $\texttt{min}_\Psi(\texttt{t}_1, \texttt{t}_2)$*: lowest value of all data points in Ψ'*
3. $\texttt{avg}_\Psi(\texttt{t}_1, \texttt{t}_2)$*: arithmetic mean of all data point values in Ψ'*
4. $\texttt{count}_\Psi(\texttt{t}_1, \texttt{t}_2)$*: number of data points in Ψ'*
5. $\texttt{sum}_\Psi(\texttt{t}_1, \texttt{t}_2)$*: sum of all data point values in Ψ'*
6. $\texttt{integral}_\Psi(\texttt{t}_1, \texttt{t}_2)$*: definite integral of the discrete function modelled by Ψ'*
7. $\texttt{stddev}_\Psi(\texttt{t}_1, \texttt{t}_2)$*: population standard deviation of all data point values in Ψ'*

Lastly, temporal aggregates summarize intervals along their temporal dimension, i.e., they operate on their length instead of the data point values residing in them. Definition 4.4 depicts the temporal aggregates supported by DTSQL.

Definition 4.4 (Temporal Aggregates) *Let Ψ be a time series, $\Pi = \{{}_{i_1}\pi_{j_1}, {}_{i_2}\pi_{j_2}, \dots\}$ a set of intervals over Ψ and $s \in \mathbb{R}$ a sample. Furthermore, let $u \in \{\texttt{weeks}, \texttt{days}, \texttt{hours}, \texttt{minutes}, \texttt{seconds}, \texttt{millis}\}$ be a time unit. Then, s is a* temporal aggregate *if it was defined using one of the following function symbols:*

1. $\texttt{max\_t}_\Psi(u, \Pi)$*: length of the longest interval in Π in unit u*
2. $\texttt{min\_t}_\Psi(u, \Pi)$*: length of the shortest interval in Π in unit u*
3. $\texttt{avg\_t}_\Psi(u, \Pi)$*: average length of the intervals in Π in unit u*
4. $\texttt{count\_t}_\Psi(\Pi)$*: number of intervals in Π in unit u*
5. $\texttt{sum\_t}_\Psi(u, \Pi)$*: sum of interval lengths in Π in unit u*
6. $\texttt{stddev\_t}_\Psi(u, \Pi)$*: population standard deviation of interval lengths in Π in unit u*

After having defined all aggregate functions supported by DTSQL, Definition 4.5 specifies the samples component of DTSQL queries.

Definition 4.5 (DTSQL Samples Component) *Let κ be a query, Ψ a time series and $S = \{s_1, s_2, \dots\}$ a set of aggregate function instances. Then, the* samples component *$\mathcal{S}$ of κ is defined by*

$$\mathcal{S} := S \tag{4.3}$$

At this point, we also introduce a notational convention for the sake of convenience. When reusing (referencing) samples in filter or event definitions, their symbolic representations are implicitly substituted by their respective value as real number. For instance, a global aggregator $\texttt{avg}_\Psi$ which is used as argument to a function in the events component is to be implicitly understood as defined in Definition 4.19 (Semantics of DTSQL Samples Component). This shorthand is warranted since it does not obscure the intended meaning, and also simplifies dealing with samples in the way they are meant to.

A concluding example of a samples component that utilizes all three kinds of aggregates is shown in Example 4.1.

Example 4.1 (DTSQL Samples Component) *Let Ψ be a time series and κ a DTSQL query that refers to the global average, a local integral between $\texttt{t}_1$ and $\texttt{t}_4$, and the total duration of the intervals $\Pi = \{{}_3\pi_8, {}_{10}\pi_{12}, {}_{15}\pi_{23}\}$ in minutes. Then, the samples component $\mathcal{S}$ of κ is*

$$\mathcal{S} = \{\texttt{max}_\Psi, \texttt{integral}_\Psi(\texttt{t}_1, \texttt{t}_4), \texttt{sum\_t}_\Psi(\texttt{minutes}, \Pi)\} \tag{4.4}$$

△

4.1.3 Filters

A *filter* enables query creators to exclude data points of an input time series Ψ from the query evaluation. Filters are applied to all data points $p \in \Psi$, *after* the computation of samples. This is important because, otherwise, sample values could not be referenced in filter definitions. Filters are conceptually simple in that they do not apply complex transformations to an input signal as, for instance, a low-pass filter does. They merely decide whether a data point p should be included in the query evaluation, solely based on the information provided by p itself. Before specifying the concrete filter function symbols supported by DTSQL, Definition 4.6 captures their common properties.

Definition 4.6 (DTSQL Filter Properties) *Let* $\mathtt{f}_p(\cdot, \ldots, \cdot)$ *be an arbitrary, n-ary filter function. Then, the following properties pertain to* $\mathtt{f}_p$*:*

1. $\mathtt{f}_p$ *is an n-ary predicate, i.e.,*

$$\forall p \in \Psi \colon \mathtt{f}_p(\cdot, \ldots, \cdot) \in \{\mathtt{true}, \mathtt{false}\} \tag{4.5}$$

2. *Since* $\mathtt{f}_p$ *may be interpreted as propositional atom, its conceptual complement is implicitly given by its logical negation. In other words, no separate filter function is required to cover the opposite meaning of* $\mathtt{f}_p$*.*

3. *If* $\mathtt{f}_p$ *is satisfied for a data point p—i.e., if* $\mathtt{f}_p(\cdot, \ldots, \cdot) = \mathtt{true}$ *holds—, then p is eligible to be included in the query evaluation.*

DTSQL provides three different kinds of filter functions: *threshold filters*, *deviation filters* and *temporal filters*. They are introduced in more detail in Definition 4.7.

Definition 4.7 (DTSQL Filter Functions) *Let* Ψ *be a time series and* $p \in \Psi$ *a data point from that time series. Then, the paragraphs below enumerate supported* filter function *instances based on p.*

Threshold Filters

- $\mathtt{lt}_p(t)$, $t \in \mathbb{R}$ *threshold:*
  Admits p to query evaluation if and only if its value component is less than the threshold t.

- $\mathtt{gt}_p(t)$, $t \in \mathbb{R}$ *a threshold:*
  Admits p to query evaluation if and only if its value component is greater than the threshold t.

Deviation Filters

- $\texttt{around\_abs}_p(r, d)$, $r \in \mathbb{R}$ *reference*, $d \in \mathbb{R}_{\geq 0}$ *maximum absolute deviation*
  Admits p to query evaluation if and only if the absolute difference between its value component and the reference value is less than the maximum deviation.

- $\texttt{around\_rel}_p(r, d)$, $r \in \mathbb{R} \setminus \{0\}$ *reference*, $d \in \mathbb{R}_{\geq 0}$ *maximum relative deviation*
  Admits p to query evaluation if and only if the relative (percentage-wise) difference between its value component and the reference value is less than the maximum deviation.

Temporal Filters

- $\texttt{before}_p(\texttt{t})$, $\texttt{t} \in \mathbb{Z}$ *a point in time (temporal bound)*
  Admits p to query evaluation if and only if its time component is before the temporal bound.

- $\texttt{after}_p(\texttt{t})$, $\texttt{t} \in \mathbb{Z}$ *a point in time (temporal bound)*
  Admits p to query evaluation if and only if its time component is after the temporal bound.

Recall that samples represent real, scalar values. Therefore, it is valid to reuse sample definitions as filter parameters whose domain are the real numbers, e.g., a threshold. Since the available filter functions have now been introduced, Definition 4.8 specifies the structure of the filter component in a DTSQL query.

Definition 4.8 (DTSQL Filter Component) *Let κ be a query, Ψ a time series, $\mathcal{F} = \{f_1, f_2, \dots\}$ a set of filter instances over Ψ and φ a* propositional formula *with $\mathcal{F}$ as set of atoms. Then, the* filter component *Φ of k is defined as the pair*

$$\Phi := (\mathcal{F}, \varphi) \tag{4.6}$$

A data point $p \in \Psi$ is included in the query evaluation if and only if the formula φ evaluates to `true` *in the interpretation given by the set of propositional atoms $\mathcal{F}$.*

The filtered time series that results from applying Φ to Ψ is denoted by $\overline{\Psi}$.

Finally, Example 4.2 demonstrates a filter component that excludes data points based on a temporal lower bound and a maximum relative deviation.

Example 4.2 (DTSQL Filter Component) *Let Ψ be a time series and κ a DTSQL query. The query should only take into account data points from Ψ which were recorded after $\texttt{t}_{11}$ and whose value components are* not *within a 5.25 % margin of the global average.*

The corresponding filter component Φ *of* κ *is*

$$\begin{aligned}\Phi &= (\mathcal{F}, \varphi)\\ &= (\{f_1, f_2\},\ (f_1 \land \neg f_2))\\ &= (\{\underbrace{\texttt{after}_p(\texttt{t}_{11})}_{f_1},\ \underbrace{\texttt{around\_rel}_p(\texttt{avg}_{\Psi}, 5.25)}_{f_2}\},\ (f_1 \land \neg f_2))\end{aligned} \tag{4.7}$$

△

4.1.4 Events

An *event* is a declarative specification of a set of intervals in which some condition(s) hold(s). In other words, they are used to detect periods where incidents that are of interest or to be further examined occur. Events are evaluated *after* filter application. In fact, they extend the notion of filters by considering detected intervals as continuous series of data points for which a propositional formula with event functions as atoms is satisfied (instead of, as with filters, examining data points only individually).

DTSQL supports two types of event functions, *filter events* and *complex events*. While filter events are syntactically and semantically equivalent to filter functions with respect to $\overline{\Psi}$, complex event functions are broader and, typically, conceptually more involved. The syntax of both is depicted in Definition 4.9

Definition 4.9 (Syntax of DTSQL Event Functions) *Let* Ψ *be a time series and* $\overline{\Psi}$ *the filtered time series as a result of applying the filter component of a query* κ*. Then, the paragraphs below enumerate supported* event function *instances.*

Filter Events
All filter functions introduced in Definition 4.7 can be used as event functions verbatim. In this context, they represent the largest possible intervals over $\overline{\Psi}$ *in which the corresponding function continuously evaluates to* `true`.

Complex Events

- `const`(s, d), $s \in \mathbb{R}_{\geq 0}$ *maximum slope,* $d \in \mathbb{R}_{\geq 0}$ *maximum relative deviation*
  Detects intervals over $\overline{\Psi}$ *with (approximately) constant values. This means the slope's absolute value of the regression line in such an interval is not greater than* s *%, and the values deviate not more than* d *% from the interval's average.*

- `increase`(l, u, t), $l, u \in \mathbb{R}_{\geq 0}$ $(u \geq l)$ *min and max change,* $t \in \mathbb{R}_{\geq 0}$ *tolerance*
  Detects intervals representing an (approximately) monotonic increase of at least l *%, but not more than* u *%. Temporary decreases are tolerated as long as their respective instantaneous rate of change does not fall below* $-t$ *%.*

- `decrease`(l, u, t), $l, u \in \mathbb{R}_{\geq 0}$ $(u \geq l)$ *min and max change,* $t \in \mathbb{R}_{\geq 0}$ *tolerance*
  Detects intervals representing an (approximately) monotonic decrease of at

least l %, but not more than u %. Temporary increases are tolerated as long as their respective instantaneous rate of change does not exceed $+t$ %.

Remember that this subsection merely serves the purpose of introducing the syntax of events. The concrete meaning of the event functions above, especially the complex ones, will be explained and exemplified in Section 4.2.4. Nevertheless, we now have gathered the definitions required to formalize the events component of a DTSQL query in Definition 4.10.

Definition 4.10 (DTSQL Events Component) *Let Ψ be a time series and κ a query. Then, the events component Ξ of κ is defined as*

$$\Xi \coloneqq \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\} \tag{4.8}$$

with the event definition *ε_i being the quintuple*

$$\varepsilon_i \coloneqq (\mathcal{E}_i,\ \chi_i,\ l_i,\ u_i,\ n_i), \ \ \textit{with } 1 \leq i \leq n \tag{4.9}$$

where $\mathcal{E}_i \coloneqq \{e_{i,1}, e_{i,2}, \ldots\}$ is a set of event function instances and χ_i a propositional formula with $\mathcal{E}_i$ as atoms. Furthermore, $l_i \in \mathbb{R}_{\geq 0}$ and $u_i \in \mathbb{R}_{\geq 0}$ with $l_i \leq u_i$ declare minimum and maximum duration constraints, *respectively. They both are expressed in the unit $n_i \in \{\texttt{weeks}, \texttt{days}, \texttt{hours}, \texttt{minutes}, \texttt{seconds}, \texttt{millis}\}$.*

Finally, Example 4.3 demonstrates the specification of an events component with multiple event definitions.

Example 4.3 (DTSQL Events Component) *Let Ψ be a time series and κ a query. It should detect two kinds of incidents:*

1. *Periods in which the recorded value is either above 300 or below 100 for at least 30 minutes.*

2. *Periods of rapid, steep monotonic increases of at least 50 % in the space of 45 seconds with a tolerance against decreases of 5.75 %. Also, such periods should not appear before the point in time $\texttt{t}_{23}$.*

The first event can be defined as

$$\begin{aligned}\varepsilon_1 &= (\mathcal{E}_1,\ \chi_1,\ l_1,\ u_1,\ n_1)\\ &= (\{e_{1,1}, e_{1,2}\},\ (e_{1,1} \vee e_{1,2}),\ 30,\ \infty,\ \texttt{minutes})\\ &= (\{\underbrace{\texttt{gt}_p(300)}_{e_{1,1}}, \underbrace{\texttt{lt}_p(100)}_{e_{1,2}}\},\ (e_{1,1} \vee e_{1,2}),\ 30,\ \infty,\ \texttt{minutes})\end{aligned} \tag{4.10}$$

The second event may be expressed similarly:

$$\begin{aligned}\varepsilon_2 &= (\mathcal{E}_2,\ \chi_2,\ l_2,\ u_2,\ n_2)\\ &= (\{e_{2,1}, e_{2,2}\},\ (e_{2,1} \land \neg e_{2,2}),\ 0,\ 45,\ \texttt{seconds})\\ &= (\{\underbrace{\texttt{increase}(50, \infty, 5.75)}_{e_{2,1}}, \underbrace{\texttt{before}_p(\texttt{t}_{23})}_{e_{2,2}}\},\ (e_{2,1} \land \neg e_{2,2}),\ 0,\ 45,\ \texttt{seconds})\end{aligned} \tag{4.11}$$

In summary, the events component of κ is

$$\Xi = \{\varepsilon_1, \varepsilon_2\} \tag{4.12}$$

△

4.1.5 Selection

The optional *selection* component of a DTSQL query κ utilizes *composition operators* to define temporal relations between the intervals represented by the events component. This makes it possible to create more complex, composite events by connecting the previously defined (atomic) events temporally.

A description of various temporal relations between intervals has already been provided in Section 2.2 (Event Detection in Time Series Data) in the form of Allen's interval algebra. In fact, DTSQL supports four of the thirteen relations introduced by [35]: `meets`, `met-by`, `before` and `after`. This is realized by means of two composition operators *precedes* and *follows* which have an optional time-gap constraint. If this constraint is present, `meets`/`met-by` semantics are used, otherwise `before`/`after`.

An instance of such an operator forms a binary event sequence. DTSQL supports n-ary sequences as well by allowing composition operators to be declared recursively. This is specified in Definition 4.11.

Definition 4.11 (Composition Operators) *Let κ be a DTSQL query with events component $\Xi = \{\varepsilon_1, \ldots, \varepsilon_n\}$. Further, let* `precedes` *and* `follows` *be the supported composition operator symbols. Then, valid composition operator instances ω with $op \in \{\texttt{precedes}, \texttt{follows}\}$ are defined as*

$$\omega = op(\boxplus_1, \boxplus_2) \qquad \textit{and} \qquad \omega = op(\boxplus_1, \boxplus_2, l, u, n) \tag{4.13, 4.14}$$

where $\boxplus_1$ and $\boxplus_2$ are either event definitions $\varepsilon \in \Xi$ or nested (recursive) composition operator instances $\widetilde{\omega}$ with $\widetilde{op} \in \{\texttt{precedes}, \texttt{follows}\}$. In Equation (4.13), no time-gap constraint is present, i.e., there should be no data points between the events specified by $\boxplus_1$ and $\boxplus_2$. In Equation (4.14), the time-gap constraint defined by $l, u \in \mathbb{R}_{\geq 0}$ $(l \geq u)$ and $n \in \{\texttt{weeks}, \texttt{days}, \texttt{hours}, \texttt{minutes}, \texttt{seconds}, \texttt{millis}\}$ represents the minimum and/or maximum duration between the intervals specified by $\boxplus_1$ and $\boxplus_2$.

If there is no recorded data point between the end of one and start of another event, then we also say one event *immediately follows/precedes* the other. Note that this does not necessarily make a point about the duration between the events—this is entirely dependent on the resolution of the data, e.g., the sampling rate.

Based on this definition, Definition 4.12 depicts the syntax of a query's selection component which, if present, consists of composition operators.

Definition 4.12 (DTSQL Selection Component) *Let κ be a DTSQL query. Then, its selection component Ω has two possible forms:*

$$\Omega \coloneqq \square \qquad \textit{and} \qquad \Omega \coloneqq \omega \tag{4.15, 4.16}$$

In Equation (4.15), *the symbol $\square$ represents "nothing", meaning no selection is present and the set of intervals detected by κ depends entirely on its events component. In Equation* (4.16), *on the other hand, ω is an arbitrarily nested composition operator instance as per Definition 4.11, representing composite events.*

An illustration of multiple event sequences, given a specific set of intervals represented by concrete event definitions, is displayed in Example 4.4.

Example 4.4 (DTSQL Selection Component) *Assume a query κ with events component $\Xi = \{\varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4, \dots\}$. Furthermore, assume three different scenarios where the evaluation of Ξ according to Section 4.2.4 yields event sequences depicted in Figure 4.1. In scenario (a), ε_1 immediately follows ε_2, and ε_4 immediately follows ε_1. In scenario (b), ε_2 is immediately followed by some unspecified event definitions ("..."), but also by ε_1 after 40 minutes, which is immediately followed by ε_4. Lastly, scenario (c) shows a simple ternary event sequence where ε_1 immediately follows ε_3 and also immediately precedes ε_4.*

$$\begin{tikzcd}
\varepsilon_2 \arrow[r] & \varepsilon_1 \arrow[r] & \varepsilon_4
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\varepsilon_2 \arrow[r] \arrow[rr, bend left, "40 \text{ min}"] & \cdots \arrow[r] & \varepsilon_1 \arrow[r] & \varepsilon_4
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
\varepsilon_3 \arrow[r] & \varepsilon_1 \arrow[r] & \varepsilon_4
\end{tikzcd}$$

(c)

Figure 4.1: Detected Event Sequences for Example 4.4

Now, consider four different selection components:

$$\begin{aligned}
\Omega_1 &= \texttt{follows}(\varepsilon_4,\, \texttt{follows}(\varepsilon_1,\, \varepsilon_2)) \\
\Omega_2 &= \texttt{follows}(\varepsilon_4,\, \texttt{follows}(\varepsilon_1,\, \varepsilon_2,\, 0,\, 45,\, \texttt{minutes})) \\
\Omega_3 &= \texttt{follows}(\varepsilon_4,\, \varepsilon_1) \\
\Omega_4 &= \texttt{precedes}(\varepsilon_2,\, \texttt{follows}(\varepsilon_4,\, \varepsilon_1),\, 0,\, 30,\, \texttt{minutes})
\end{aligned} \tag{4.17}$$

Ω_1 represents scenario (a), but neither (b) nor (c). Ω_2 describes scenario (b), and maybe (a)—depending on the time between ε_2 and ε_1. Ω_3 characterizes all scenarios (a), (b) and (c). Finally, Ω_4 is not guaranteed to match any scenario—only (a) could potentially fit, if the end of ε_2 and the start of ε_1 are not more than 30 minutes apart. △

4.1.6 Yield

The *yield* statement ultimately determines the result of a DTSQL query. DTSQL supports six different *yield* formats which are explained in Definition 4.13.

Definition 4.13 (Yield Formats) *The supported yield formats of a DTSQL query κ are defined by the set*

$$\mathbb{Y} := \{\texttt{allints}, \texttt{maxints}, \texttt{minints}, \texttt{datapoints}, \texttt{sample}, \texttt{sampleset}\} \tag{4.18}$$

The element `allints` *represents all intervals captured by κ's events and/or selection component. The longest and shortest intervals are addressed using* `maxints` *and* `minints`*, respectively. With* `datapoints`*, all data points that have not been filtered out or—if an events or selection component is present—are part of a detected period are returned. Finally,* `sample` *and* `sampleset` *are used to return the concrete (computed) value(s) of one or multiple sample(s).*

The latter two yield formats—`sample` and `sampleset`—need to be combined with a *result definition parameter*. This parameter denotes the sample(s) to be returned. This knowledge makes it possible to specify the syntax of the yield component in Definition 4.14.

Definition 4.14 (DTSQL Yield Component) *Let κ be a DTSQL query with samples component $\mathcal{S}$. Its yield component is defined as the pair*

$$\Upsilon := (\gamma, d) \tag{4.19}$$

with yield format $\gamma \in \mathbb{Y}$ and result definition parameter

$$d = \begin{cases} S \subseteq \mathcal{S}\ (S \neq \emptyset), & \textit{if } \gamma = \texttt{samples} \\ s \in \mathcal{S}, & \textit{if } \gamma = \texttt{sample} \\ \square, & \textit{else} \end{cases} \tag{4.20}$$

where the symbol $\square$ represents "nothing" (no parameter).

An exhaustive showcase of all supported yield formats is presented in Example 4.5.

Example 4.5 (DTSQL Yield Component) *Let κ be a query with samples component $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$. Then, possible yield components include:*

$$
\begin{aligned}
\Upsilon_1 &= (\texttt{allints}, \square)\\
\Upsilon_2 &= (\texttt{maxints}, \square)\\
\Upsilon_3 &= (\texttt{minints}, \square)\\
\Upsilon_4 &= (\texttt{datapoints}, \square)\\
\Upsilon_5 &= (\texttt{sample}, s_2)\\
\Upsilon_6 &= (\texttt{samples}, \{s_1, s_3, s_4\})
\end{aligned}
\tag{4.21}
$$

△

4.2 Language Semantics

This section specifies the semantics of a DTSQL query, i.e., how the result is determined. At first, an intuition for the evaluation of a query is provided by succinctly computing the result of an example query step-by-step over a small input time series. Afterwards, subsections each dedicated to one of the query components introduced in Section 4.1 thoroughly explain and specify their semantics.

4.2.1 DTSQL Query

The result of a DTSQL query is determined by successively evaluating its individual components. This is a five-step process:

1. Compute samples as described in Section 4.2.2.
2. Filter out data points violating the filter definition as defined in Section 4.2.3.
3. Detect intervals corresponding to the event definitions according to Section 4.2.4.
4. Relate detected intervals to each other temporally according to the selection component as per Section 4.2.5.
5. Assemble the final result based on the yield statement as presented in Section 4.2.6.

Formally, since the semantics of the individual components build upon each other, the result of a query is equivalent to the evaluation of its yield component. This is also stated in Definition 4.15. The actual meaning of this definition will be clear after reading the remaining subsection regarding query semantics.

Definition 4.15 (Semantics of DTSQL Query) *Let Ψ be a time series and $\kappa = (\mathcal{S}, \Phi, \Xi, \Omega, \Upsilon)$ a DTSQL query over the input time series Ψ. Then, the result of κ is denoted as and defined to be equivalent to*

$$\texttt{res}(\kappa, \Psi), \tag{4.22}$$

where the function `res` is defined as presented in Definition 4.28 (Semantics of DTSQL Yield Component).

As mentioned in the introduction of this section, Example 4.6 provides a step-by-step result computation of a simple query on a small time series. The goal of this example is not formal rigor and precision, but to give an intuition as to how DTSQL queries are evaluated. Therefore, the individual query components are evaluated rather concisely. Again, for explanations on the notations used throughout the example, refer to the respective component's subsection.

Example 4.6 (Semantics of DTSQL Query) *Suppose there is an input time series $\Psi = \langle p_1, \ldots, p_{10} \rangle$ as defined by Table 4.1. Cut off all data points whose values are within a ± 25 % range of the global maximum. Then, find intervals where values are consistently below and above the global average, respectively. Relate such intervals to each other so that those with values less than the average appear before those with values above the average. Finally, return all data points that were recorded during such an event sequence.*

data point	time	identifier	value
p_1	`2022-08-27 12:00:00`	t₁	1
p_2	`2022-08-27 12:15:00`	t₂	2
p_3	`2022-08-27 12:30:00`	t₃	3
p_4	`2022-08-27 12:45:00`	t₄	4
p_5	`2022-08-27 13:00:00`	t₅	5
p_6	`2022-08-27 13:15:00`	t₆	6
p_7	`2022-08-27 13:30:00`	t₇	7
p_8	`2022-08-27 13:45:00`	t₈	8
p_9	`2022-08-27 14:00:00`	t₉	9
p_{10}	`2022-08-27 14:15:00`	t₁₀	10

Table 4.1: Exemplary Time Series for Example 4.6

A DTSQL query expressing these criteria is:

$$\begin{aligned}
\kappa &= (\mathcal{S}, \Phi, \Xi, \Omega, \Upsilon) \\
&= \Big(\{s_1\}, \\
&\quad (\{f_1\}, \neg f_1), \\
&\quad \{(\{e_{1,1}, e_{1,1}, 0, \infty, \texttt{minutes}\}), (\{e_{2,1}, e_{2,1}, 0, \infty, \texttt{minutes}\})\}, \\
&\quad \texttt{precedes}(\varepsilon_1, \varepsilon_2), \\
&\quad (\texttt{datapoints}, \Box)\Big) \\
&= \Big(\{\texttt{avg}_\Psi\}, \\
&\quad (\{\underbrace{\texttt{around\_rel}_p(\texttt{avg}_\Psi, 25)}_{f_1}\}, \neg f_1), \\
&\quad \{\underbrace{(\{\texttt{lt}_p(\texttt{avg}_\Psi)\}, e_{1,1}, 0, \infty, \texttt{minutes})}_{\varepsilon_1}, \underbrace{(\{\texttt{gt}_p(\texttt{avg}_\Psi)\}, e_{2,1}, 0, \infty, \texttt{minutes})}_{\varepsilon_2}\}, \\
&\quad \texttt{precedes}(\varepsilon_1, \varepsilon_2), \\
&\quad (\texttt{datapoints}, \Box)\Big)
\end{aligned} \tag{4.23}$$

In order to evaluate κ, we go through the five steps outlined above:

1. *The set of computed samples amounts to*

$$\mathcal{S}^c = \{\texttt{eval\_s}(\texttt{avg}_\Psi)\} = \{5.5\} \tag{4.24}$$

2. *The range* $\texttt{avg}_\Psi \pm 25\ \%$ *is equivalent to* $5.5 \pm 25\ \% = [4.125, 6.875]$. *Therefore, we have* $\texttt{eval}_{I_p}(\neg f_1) = \texttt{true}$ *for* $p \in \{p_1, p_2, p_3, p_4, p_7, p_8, p_9, p_{10}\}$ *and* $\texttt{eval}_{I_p}(\neg f_1) = \texttt{false}$ *for* $p \in \{p_5, p_6\}$. *Hence, the filtered time series is equal to*

$$\overline{\Psi} = \{p_1, p_2, p_3, p_4, p_7, p_8, p_9, p_{10}\} \tag{4.25}$$

3. *The intervals represented by the two event definitions ε_1 and ε_2 are evident from Table 4.1. We have* $\texttt{ints}(\varepsilon_1) = \{{}_1\pi_4\}$ *and* $\texttt{ints}(\varepsilon_2) = \{{}_7\pi_{10}\}$, *which leads to*

$$\texttt{ints}(\Xi) = \{(\varepsilon_1, \{{}_1\pi_4\}), (\varepsilon_2, \{{}_7\pi_{10}\})\} \tag{4.26}$$

4. *The selection component Ω defines composite events where an interval characterized by ε_1 appears* before *an interval characterized by ε_2. When examining Equation* (4.26)*, it is clear that*

$$\texttt{ints}(\Omega) = \{{}_1\pi_{10}\} \tag{4.27}$$

5. *The yield component asks for all data points contained in the intervals from the composition operator, as shown in Equation* (4.27)*. This also resorts back to the filtered time series in Equation* (4.25)*. Therefore, the final result of the query is*

$$\begin{aligned}\texttt{res}(\kappa, \Psi) &= \bigcup_{{}_i\pi_j \in \texttt{ints}(\Omega)} \texttt{ps}_{\overline{\Psi}}({}_i\pi_j) \\ &= \texttt{ps}_{\overline{\Psi}}({}_1\pi_{10}) \\ &= \{p_1, p_2, p_3, p_4, p_7, p_8, p_9, p_{10}\}\end{aligned} \tag{4.28}$$

△

4.2.2 Samples

As already outlined by the syntax definition of samples in Section 4.1.2, DTSQL supports three types of samples—global value aggregates, local value aggregates and temporal aggregates. The following paragraphs provide a precise specification of the values represented by the them.

The global value aggregates, as shown in Definition 4.16, follow standard definitions of maximum, minimum, arithmetic mean and population standard deviation. Note that the integral in Equation (4.34) is only a slight modification of Definition 2.10 (Integral of a Time Series). It explicitly converts the time difference between adjacent data points to seconds so that the integral is calculated using the SI base unit of time.

Definition 4.16 (Semantics of Global Value Aggregates) *Let* $\Psi = \langle p_1, \ldots, p_n \rangle$ *be a time series and* `eval_s`(s) *a function that maps an aggregator instance* s *to a real value. Then, the global value aggregate functions of* DTSQL *are defined as follows:*

$$\texttt{eval\_s}(\texttt{max}_\Psi) \coloneqq \max_{p \in \mathcal{P}(\Psi)} \texttt{dpv}(p) \tag{4.29}$$

$$\texttt{eval\_s}(\texttt{min}_\Psi) \coloneqq \min_{p \in \mathcal{P}(\Psi)} \texttt{dpv}(p) \tag{4.30}$$

$$\texttt{eval\_s}(\texttt{avg}_\Psi) \coloneqq \frac{\texttt{eval\_s}(\texttt{sum}_\Psi)}{\texttt{eval\_s}(\texttt{count}_\Psi)} \tag{4.31}$$

$$\texttt{eval\_s}(\texttt{count}_\Psi) \coloneqq |\Psi| \tag{4.32}$$

$$\texttt{eval\_s}(\texttt{sum}_\Psi) \coloneqq \sum_{p \in \mathcal{P}(\Psi)} (\texttt{dpv}(p)) \tag{4.33}$$

$$\texttt{eval\_s}(\texttt{integral}_\Psi) \coloneqq \frac{1}{2} \cdot \sum_{i=1}^{n-1} \Big(\big(\texttt{dpv}(p_i) + \texttt{dpv}(p_{i+1})\big) \cdot \texttt{duration}\big(\texttt{dpt}(p_{i+1}) - \texttt{dpt}(p_i), \texttt{seconds}\big) \Big) \tag{4.34}$$

$$\texttt{eval\_s}(\texttt{stddev}_\Psi) \coloneqq \sqrt{\frac{\mathit{squares}}{\texttt{eval\_s}(\texttt{count}_\Psi)}}, \quad \mathit{with\ squares} = \sum_{p \in \mathcal{P}(\Psi)} \Big(\texttt{dpv}(p) - \texttt{eval\_s}(\texttt{avg}_\Psi)\Big)^2 \tag{4.35}$$

In order to keep the definitions simple, the semantics of local value aggregates are expressed as a reduction to global value aggregates on a temporally constrained variant of the input time series. This is formally defined in Definition 4.17.

Definition 4.17 (Semantics of Local Value Aggregates) *Let Ψ be a time series and let $\mathtt{t}_1, \mathtt{t}_2 \in \mathbb{Z}$ with $\mathtt{t}_2 \geq \mathtt{t}_1$ be two arbitrary points in time. Then, Ψ' is the order-preserving variant of Ψ containing only data points whose time components are between $\mathtt{t}_1$ and $\mathtt{t}_2$, such that*

$$\mathcal{P}(\Psi') = \{p \in \Psi \mid \texttt{dpt}(p) \geq \mathtt{t}_1 \land \texttt{dpt}(p) \leq \mathtt{t}_2\} \tag{4.36}$$

holds. Further, let `eval_s`(s) *be a function that maps an aggregator instance s to a real value. Then, the local value aggregates of DTSQL are evaluated as follows:*

$$\texttt{eval\_s}(\texttt{max}_\Psi(\mathtt{t}_1, \mathtt{t}_2)) \coloneqq \texttt{eval\_s}(\texttt{max}_{\Psi'}) \tag{4.37}$$

$$\texttt{eval\_s}(\texttt{min}_\Psi(\mathtt{t}_1, \mathtt{t}_2)) \coloneqq \texttt{eval\_s}(\texttt{min}_{\Psi'}) \tag{4.38}$$

$$\texttt{eval\_s}(\texttt{avg}_\Psi(\mathtt{t}_1, \mathtt{t}_2)) \coloneqq \texttt{eval\_s}(\texttt{avg}_{\Psi'}) \tag{4.39}$$

$$\texttt{eval\_s}(\texttt{count}_\Psi(\mathtt{t}_1, \mathtt{t}_2)) \coloneqq \texttt{eval\_s}(\texttt{count}_{\Psi'}) \tag{4.40}$$

$$\texttt{eval\_s}(\texttt{sum}_\Psi(\mathtt{t}_1, \mathtt{t}_2)) \coloneqq \texttt{eval\_s}(\texttt{sum}_{\Psi'}) \tag{4.41}$$

$$\texttt{eval\_s}(\texttt{integral}_\Psi(\mathtt{t}_1, \mathtt{t}_2)) \coloneqq \texttt{eval\_s}(\texttt{integral}_{\Psi'}) \tag{4.42}$$

$$\texttt{eval\_s}(\texttt{stddev}_\Psi(\mathtt{t}_1, \mathtt{t}_2)) \coloneqq \texttt{eval\_s}(\texttt{stddev}_{\Psi'}) \tag{4.43}$$

Ultimately, the semantics of temporal aggregates are very similar to the ones of global aggregates and are depicted in Definition 4.18. Note that there is no temporal aggregator for the calculation of an integral because that operation is not applicable in this context.

Definition 4.18 (Semantics of Temporal Aggregates) *Let Ψ be a time series and $\Pi = \{{}_{i_1}\pi_{j_1}, {}_{i_2}\pi_{j_2}, \dots\}$ a set of intervals over Ψ and $u \in \{\texttt{weeks}, \texttt{days}, \texttt{hours}, \texttt{minutes}, \texttt{seconds}, \texttt{millis}\}$ a time unit. Furthermore, let*

$$\delta\colon \Pi \to \mathbb{R},\ {}_i\pi_j \mapsto \texttt{duration}(|{}_i\pi_j|, u) \tag{4.44}$$

be a function that maps an interval to its duration in the unit u and $\texttt{eval\_s}(s)$ a function that maps an aggregator instance s to a real value. Then, the temporal aggregate functions supported by DTSQL are defined as follows:

$$\texttt{eval\_s}(\texttt{max\_t}_\Psi(u, \Pi)) \coloneqq \max_{{}_i\pi_j \in \Pi} \delta({}_i\pi_j) \tag{4.45}$$

$$\texttt{eval\_s}(\texttt{min\_t}_\Psi(u, \Pi)) \coloneqq \min_{{}_i\pi_j \in \Pi} \delta({}_i\pi_j) \tag{4.46}$$

$$\texttt{eval\_s}(\texttt{avg\_t}_\Psi(u, \Pi)) \coloneqq \frac{\texttt{eval\_s}(\texttt{sum\_t}_\Psi(u, \Pi))}{\texttt{eval\_s}(\texttt{count\_t}_\Psi(\Pi))} \tag{4.47}$$

$$\texttt{eval\_s}(\texttt{count\_t}_\Psi(\Pi)) \coloneqq |\Pi| \tag{4.48}$$

$$\texttt{eval\_s}(\texttt{sum\_t}_\Psi(u, \Pi)) \coloneqq \sum_{{}_i\pi_j \in \Pi} (\delta({}_i\pi_j)) \tag{4.49}$$

$$\begin{aligned} \texttt{eval\_s}(\texttt{stddev\_t}_\Psi(u, \Pi)) &\coloneqq \sqrt{\frac{\mathit{squares}}{\texttt{eval\_s}(\texttt{count\_t}_\Psi(\Pi))}}, \\ \textit{with squares} &= \sum_{{}_i\pi_j \in \Pi} \Big(\delta({}_i\pi_j) - \texttt{eval\_s}(\texttt{avg\_t}_\Psi(u, \Pi))\Big)^2 \end{aligned} \tag{4.50}$$

Building upon these definitions, the overall semantics of a samples component is summarized in Definition 4.19.

Definition 4.19 (Semantics of DTSQL Samples Component) *Let Ψ be a time series and κ a query with samples component $\mathcal{S} = \{s_1, s_2, \dots\}$. Then, the set of sample values—i.e., the evaluated samples component—$\mathcal{S}^c \subset \mathbb{R}$ is defined as:*

$$\mathcal{S}^c \coloneqq \{\texttt{eval\_s}(s) \mid s \in \mathcal{S}\} \tag{4.51}$$

In order to finalize this subsection, Example 4.7 provides an exemplary definition and evaluation of a samples component.

Example 4.7 (Semantics of DTSQL Samples Component) *Consider the time series $\Psi = \langle p_1, \dots, p_4\rangle$ depicted in Table 4.2.*

data point	time	identifier	value
p_1	`2022-08-17 12:00:00`	$\mathtt{t}_1$	15
p_2	`2022-08-17 12:23:00`	$\mathtt{t}_2$	20
p_3	`2022-08-17 12:45:00`	$\mathtt{t}_3$	14
p_4	`2022-08-07 13:02:00`	$\mathtt{t}_4$	22

Table 4.2: Exemplary Time Series for Example 4.7

Let κ be a query that captures a global minimum, local standard deviation and temporal average. More concretely, κ has a samples component

$$\mathcal{S} = \{\mathtt{min}_\Psi, \mathtt{stddev}_\Psi(\mathtt{t}_1, \mathtt{t}_3), \mathtt{avg\_t}_\Psi(\mathtt{minutes}, \underbrace{\{{}_1\pi_2, {}_2\pi_3, {}_3\pi_4\}}_{\Pi})\} \tag{4.52}$$

Then, the concrete sample values are as follows:

$$\mathtt{eval\_s}(\mathtt{min}_\Psi) = \min_{p \in \mathcal{P}(\Psi)} \mathtt{dpv}(p) = 25 \tag{4.53}$$

$$\begin{aligned} \Psi' &= \langle p_1, p_2, p_3 \rangle, \\ \mathtt{eval\_s}(\mathtt{avg}_{\Psi'}) &= \frac{28.5 + 27.24 + 30.19}{3} = 28.64\dot{3} \\ squares &= (28.5 - 28.64\dot{3})^2 + (27.24 - 28.64\dot{3})^2 \\ &\quad + (30.19 - 28.64\dot{3})^2 \\ &\approx 4.3821 \\ \implies \mathtt{eval\_s}(\mathtt{stddev}_\Psi(\mathtt{t}_1, \mathtt{t}_3)) &= \mathtt{stddev}_{\Psi'} \\ &= \sqrt{\frac{1}{3} \cdot squares} \\ &\approx 1.21 \end{aligned} \tag{4.54}$$

$$\begin{aligned} \mathtt{eval\_s}(\mathtt{avg\_t}_\Psi(\mathtt{minutes}, \Pi)) &= \frac{1}{3} \cdot \Big(\mathtt{duration}(\mathtt{t}_2 - \mathtt{t}_1, \mathtt{minutes}) \\ &\qquad + \mathtt{duration}(\mathtt{t}_3 - \mathtt{t}_2, \mathtt{minutes}) \\ &\qquad + \mathtt{duration}(\mathtt{t}_4 - \mathtt{t}_3, \mathtt{minutes})\Big) \\ &= \frac{23 + 22 + 17}{3} \\ &\approx 20.67 \end{aligned} \tag{4.55}$$

In summary, the evaluated samples component of κ is equivalent to

$$\overline{\mathcal{S}} = \{25,\ 1.21,\ 20.67\} \tag{4.56}$$

$\triangle$

4.2.3 Filters

We now define a representation of the set of data points which are actually considered when evaluating a DTSQL query, based on the syntax of the filter component defined in Section 4.1.3. Also, recall the semantics of a propositional formula from Section 2.5.

At first, we need to specify the meaning of filter functions. As explained in Section 4.1.3, the decision whether a filter function evaluates to `true` or `false` depends on a concrete data point. Definition 4.20 provides specifications of how the filter functions supported by DTSQL are evaluated.

Definition 4.20 (Semantics of DTSQL Filter Functions) *Let Ψ be a time series, $p \in \Psi$ a data point from that time series and f a filter function from Definition 4.7. Further, let $\texttt{eval\_f}(f, p)$ be an evaluation function that determines whether p satisfies f. Then, the semantics of filter functions of DTSQL are as follows.*

Threshold Filters

$$\texttt{eval\_f}(\texttt{lt}_p(t), p) = \texttt{true} :\Longleftrightarrow \texttt{dpv}(p) < t \tag{4.57}$$

$$\texttt{eval\_f}(\texttt{gt}_p(t), p) = \texttt{true} :\Longleftrightarrow \texttt{dpv}(p) > t \tag{4.58}$$

Deviation Filters

$$\texttt{eval\_f}(\texttt{around\_abs}_p(r, d), p) = \texttt{true} :\Longleftrightarrow |\texttt{dpv}(p) - r| \leq d \tag{4.59}$$

$$\texttt{eval\_f}(\texttt{around\_rel}_p(r, d), p) = \texttt{true} :\Longleftrightarrow \frac{|\texttt{dpv}(p) - r|}{|r|} \cdot 100 \leq d \tag{4.60}$$

Temporal Filters

$$\texttt{eval\_f}(\texttt{before}_p(\texttt{t}), p) = \texttt{true} :\Longleftrightarrow \texttt{dpt}(p) < t \tag{4.61}$$

$$\texttt{eval\_f}(\texttt{after}_p(\texttt{t}), p) = \texttt{true} :\Longleftrightarrow \texttt{dpt}(p) > t \tag{4.62}$$

Based on the just determined semantics of filter functions, we are now able to specify the set of data points admitted to query evaluation—see Definition 4.21.

Definition 4.21 (Semantics of DTSQL Filter Component) *Let Ψ be a time series and κ a query with filter component $\Phi = (\mathcal{F}, \varphi)$. Moreover, let $I_p \colon \mathcal{F} \to \{\texttt{true}, \texttt{false}\}$ be an interpretation function that assigns truth values to filter function instances—the atoms of φ—, given a data point $p \in \Psi$. It is defined as follows:*

$$I_p(f) \coloneqq \texttt{eval\_f}(f, p), \text{ with } f \in \mathcal{F} \tag{4.63}$$

Then, the filtered time series $\overline{\Psi}$ contains only those data points p from Ψ which *satisfy φ in the interpretation I_p:*

$$\mathcal{P}(\overline{\Psi}) \coloneqq \{p \in \mathcal{P}(\Psi) \mid \texttt{eval}_{I_p}(\varphi) = \texttt{true}\} \tag{4.64}$$

Concluding, Example 4.8 demonstrates the evaluation of a filter component, given a concrete time series.

Example 4.8 (Semantics of DTSQL Filter Component) *Consider the time series $\Psi = \langle p_1, \dots, p_4 \rangle$ depicted in Table 4.3.*

data point	time	value
p_1	t_1	15
p_2	t_2	20
p_3	t_3	14
p_4	t_4	22

Table 4.3: Exemplary Time Series for Example 4.8

Let κ be a query with filter component $\Phi_i = (\mathcal{F}, \varphi)$ and $i \in \{1, 2, 3\}$. The trivial filters

$$\Phi_1 = (\emptyset, \texttt{true}) \tag{4.65}$$

$$\Phi_2 = (\emptyset, \texttt{false}) \tag{4.66}$$

result in $\mathcal{P}(\overline{\Psi}) = \{p_1, \dots, p_4\}$ and $\mathcal{P}(\overline{\Psi}) = \emptyset$, respectively.

In contrast to that, a filter component which only admits values that are either in the (absolute) range 22 ± 2 or less than 15, i.e.,

$$\begin{aligned} \Phi_3 &= (\mathcal{F},\ \varphi) \\ &= (\{f_1, f_2\},\ \varphi) \\ &= (\{\underbrace{\texttt{around\_abs}_p(22, 2)}_{f_1},\ \underbrace{\texttt{lt}_p(15)}_{f_2}\},\ (f_1 \vee f_2)) \end{aligned} \tag{4.67}$$

results in $\mathcal{P}(\overline{\Psi}) = \{p_2, p_3, p_4\}$ because

$$\begin{aligned} I_{p_1}(f_1) &= \texttt{eval\_f}(\texttt{around\_abs}_{p_1}(22, 2),\ (\mathsf{t}_1, 15)) = \texttt{false}\ ✗ \\ I_{p_1}(f_2) &= \texttt{eval\_f}(\texttt{lt}_{p_1}(15),\ (\mathsf{t}_1, 15)) = \texttt{false}\ ✗ \end{aligned} \tag{4.68}$$

$$I_{p_2}(f_1) = \texttt{eval\_f}(\texttt{around\_abs}_{p_2}(22, 2),\ (\mathsf{t}_2, 20)) = \texttt{true}\ ✓ \tag{4.69}$$

$$\begin{aligned} I_{p_3}(f_1) &= \texttt{eval\_f}(\texttt{around\_abs}_{p_3}(22, 2),\ (\mathsf{t}_3, 14)) = \texttt{false}\ ✗ \\ I_{p_3}(f_2) &= \texttt{eval\_f}(\texttt{lt}_{p_3}(15),\ (\mathsf{t}_3, 14)) = \texttt{true}\ ✓ \end{aligned} \tag{4.70}$$

$$I_{p_4}(f_1) = \texttt{eval\_f}(\texttt{around\_abs}_{p_4}(22, 2),\ (\mathsf{t}_4, 22)) = \texttt{true}\ ✓ \tag{4.71}$$

△

4.2.4 Events

After introducing the events component of a DTSQL query in Section 4.1.4, this section provides a precise specification of the intervals represented by the event functions supported by the language. Filter events are all evaluated the same way. The semantics of complex events, however, needs to be defined individually because, in general, they do not exhibit many similarities.

There is, however, one key commonality among all kinds of event functions. Generally, we are only interested in *maximal intervals*, i.e., ones that cannot be extended without violating the event function. For filter events, this means that extensions of the respective interval by exactly one data point to the left and right, respectively, are examined. If we were to look even further, the interval might not continuously satisfy the filter event function, and we do not consider intervals with gaps. For complex events, it may be possible to extend an interval in either direction—temporarily violating the event function—and extending it even more to, in the end, satisfy it again. In other words, extending a locally maximal interval satisfying a complex event to a globally maximal one is permissible, if temporarily invalidating it in the process is acceptable.

The evaluation of filter event function instances is specified in Definition 4.22.

Definition 4.22 (Semantics of DTSQL Filter Events) *Let $\overline{\Psi} = \langle p_1, \ldots, p_n \rangle$ be filtered a time series resulting from a DTSQL filter application, ${}_i\pi_j$ an arbitrary interval over $\overline{\Psi}$ with $i, j \in \{1, \ldots, n\}$ and e a filter event function from Definition 4.9 delegating to the filter function f. Moreover, let* $\texttt{eval\_e}(e, {}_i\pi_j)$ *be an evaluation function that determines whether ${}_i\pi_j$ satisfies e. This is the case if and only if f is continuously satisfied during the interval and ${}_i\pi_j$ is maximal.*

In more formal terms, this means $\texttt{eval\_e}(e, {}_i\pi_j)$ *is defined as*

$$\begin{aligned}\texttt{eval\_e}(e, {}_i\pi_j) = \texttt{true} \;:\Longleftrightarrow\; & (\forall p \in \texttt{ps}_{\overline{\Psi}}({}_i\pi_j)\colon \texttt{eval\_f}(f, p) = \texttt{true}) \\ & \wedge\, \texttt{eval\_f}(f, p_{i-1}) = \texttt{false} \\ & \wedge\, \texttt{eval\_f}(f, p_{j+1}) = \texttt{false}\end{aligned} \tag{4.72}$$

where $\texttt{eval}_\texttt{f}$ *is evaluted on $\overline{\Psi}$.*

The semantics of detecting intervals with constant values are given in Definition 4.23.

Definition 4.23 (Semantics of DTSQL Constant Events) *Let $\overline{\Psi} = \langle p_1, \ldots, p_n \rangle$ be filtered a time series resulting from a DTSQL filter application, and ${}_i\pi_j$ an arbitrary interval over $\overline{\Psi}$ with $i, j \in \{1, \ldots, n\}$. Moreover, let the evaluation function* $\texttt{eval\_e}$ *determine whether ${}_i\pi_j$ satisfies* $\texttt{const}(s, d)$*, denoted by the expression* $\texttt{eval\_e}(\texttt{const}(s, d), {}_i\pi_j)$*. A constant interval is defined by three criteria:*

1. *The slope of the simple linear regression line over ${}_i\pi_j$ must not exceed s %.*

2. *The value of all data points in ${}_i\pi_j$ must not deviate more than d % from their average value.*

3. *The interval ${}_i\pi_j$ is (non-locally) maximal with respect to (1) and (2).*

The first condition is equivalent to

$$\mathtt{regc}({}_i\pi_j, s) \coloneqq |\beta_1| \cdot 100 \leq s \tag{4.73}$$

where β_1 is the slope of the regression line over ${}_i\pi_j$ as defined in Definition 2.12 (Linear Regression on Time Series) with Δx being equal to the time difference between pairwise adjacent data points, expressed in the unit determined by Algorithm 2.1.

The second condition is represented by

$$\begin{aligned}\mathtt{devc}({}_i\pi_j, d) \;\coloneqq\; & \forall p \in \mathtt{ps}_{\overline{\Psi}}({}_i\pi_j)\colon \\ & \mathtt{eval\_f}(\mathtt{around\_rel}_p(\mathtt{avg}_{\overline{\Psi}}(\mathsf{t_i}, \mathsf{t_j}),\ d),\ p) = \mathtt{true}\end{aligned} \tag{4.74}$$

The maximality constraint is given as

$$\begin{aligned}\mathtt{maxc}({}_i\pi_j, s, d) \coloneqq \neg\exists a, b \in \mathbb{N}\colon\ & \neg(a = 0 \land b = 0) \\ & \land\ \mathtt{regc}({}_{i-a}\pi_{j+b}, s) = \mathtt{true} \\ & \land\ \mathtt{devc}({}_{i-a}\pi_{j+b}, d) = \mathtt{true}\end{aligned} \tag{4.75}$$

Finally, the evaluation of the complex event function instance $\mathtt{const}(s, d)$ *is given by*

$$\begin{aligned}\mathtt{eval\_e}(\mathtt{const}(s, d),\ {}_i\pi_j) = \mathtt{true} \;:\Longleftrightarrow\; & \mathtt{regc}({}_i\pi_j, s) = \mathtt{true} \\ & \land\ \mathtt{devc}({}_i\pi_j, d) = \mathtt{true} \\ & \land\ \mathtt{maxc}({}_i\pi_j, s, d) = \mathtt{true}\end{aligned} \tag{4.76}$$

The two remaining event definition functions, `increase` and `decrease`, describe intervals representing a monotonic progression. They exact meanings are similar enough to be specified togehter—see Definition 4.24.

Definition 4.24 (Semantics of DTSQL Monotonic Events) *Let $\overline{\Psi} = \langle p_1, \ldots, p_n\rangle$ be filtered a time series resulting from a DTSQL filter application, and ${}_i\pi_j$ an arbitrary interval over $\overline{\Psi}$ with $i, j \in \{1, \ldots, n\}$. Moreover, let e be a monotonic event function instance, i.e., either* $\mathtt{increase}(l, u, t)$ *or* $\mathtt{decrease}(l, u, t)$*. Furthermore, let* $\mathtt{eval\_e}(e, {}_i\pi_j)$ *be a function that determines whether ${}_i\pi_j$ satisfies e. A monotonic interval is defined by three criteria:*

1. *The relative change in value (increase or decrease, respectively) between p_i and p_j is at least l %, but not more than u %.*

2. *The instantaneous rate of change of the data points in* ${}_i\pi_j$ *never falls under* $-t$ % *or exceeds* $+t$ %*, respectively—depending on whether* e *describes a monotonic increase or decrease.*

3. *The interval* ${}_i\pi_j$ *is (non-locally) maximal with respect to (1) and (2).*

The first condition is encoded as

$$\begin{aligned}\texttt{difc}({}_i\pi_j, l, u) &:= \begin{cases}\texttt{ch}({}_i\pi_j) \geq 0 \wedge \texttt{ch}(i,j) \in [l,u], & \textit{if } e = \texttt{increase}(l,u,t)\\ \texttt{ch}({}_i\pi_j) \leq 0 \wedge |\texttt{ch}(i,j)| \in [l,u], & \textit{if } e = \texttt{decrease}(l,u,t)\end{cases},\\ \textit{with } \texttt{ch}({}_i\pi_j) &:= \frac{\texttt{dpv}(p_j) - \texttt{dpv}(p_i)}{|\texttt{dpv}(p_i)|} \cdot 100\end{aligned} \tag{4.77}$$

where $\texttt{ch}(i,j)$ *is the change in value between the bounds of* ${}_i\pi_j$ *in percent.*

The second constraint is equivalent to

$$\begin{aligned}\texttt{ratc}({}_i\pi_j, t) &:= \begin{cases}\forall \widehat{p} \in \widehat{\mathcal{P}}(i,j)\colon \texttt{dpv}(\widehat{p}) \cdot 100 \geq -t, & \textit{if } e = \texttt{increase}(l,u,t)\\ \forall \widehat{p} \in \widehat{\mathcal{P}}(i,j)\colon \texttt{dpv}(\widehat{p}) \cdot 100 \leq t & \textit{if } e = \texttt{decrease}(l,u,t)\end{cases},\\ \textit{with } \widehat{\mathcal{P}}(i,j) &:= \{\widehat{p} \in \widehat{\Psi} \mid \texttt{dpt}(\widehat{p}) \geq \mathsf{t_i} \wedge \texttt{dpt}(\widehat{p}) \leq \mathsf{t_j}\}\end{aligned} \tag{4.78}$$

where $\widehat{\Psi} = \langle \widehat{p_1}, \widehat{p_2}, \ldots, \widehat{p_n} \rangle$ *is the first derivative of* $\overline{\Psi}$ *as defined in Definition 2.8 (Derivative of a Time Series).*

The maximality constraint is given by

$$\begin{aligned}\texttt{maxc}({}_i\pi_j, l, u, t) := \neg\exists a, b \in \mathbb{N}\colon\ &\neg(a = 0 \wedge b = 0)\\ &\wedge \texttt{difc}({}_{i-a}\pi_{j+b}, l, u) = \texttt{true}\\ &\wedge \texttt{ratc}({}_{i-a}\pi_{j+b}, t) = \texttt{true}\end{aligned} \tag{4.79}$$

Finally, the evaluation of the monotonic event function instance $e \in \{\texttt{increase}(l,u,t), \texttt{decrease}(l,u,t)\}$ *is defined as*

$$\begin{aligned}\texttt{eval\_e}(e, {}_i\pi_j) = \texttt{true} :\Longleftrightarrow\ &\texttt{difc}({}_i\pi_j, l, u) = \texttt{true}\\ &\wedge \texttt{ratc}({}_i\pi_j, t) = \texttt{true}\\ &\wedge \texttt{maxc}({}_i\pi_j, l, u, t) = \texttt{true}\end{aligned} \tag{4.80}$$

Since the semantics of all event functions have now been defined, they can be used to formalize the meaning of event definitions. This is depicted in Definition 4.25.

Definition 4.25 (Semantics of DTSQL Events Component) *Let $\overline{\Psi}$ be a time series resulting from a DTSQL filter application and κ a query with an events component*

$$\varXi = \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\} \tag{4.81}$$

and event definitions

$$\varepsilon_i = (\mathcal{E}_i,\, \chi_i,\, l_i,\, u_i,\, n_i), \text{ with } 1 \leq i \leq n \tag{4.82}$$

Moreover, let $I_{{}_j\pi_k} : \mathcal{E}_i \rightarrow \{\texttt{true}, \texttt{false}\}$ be an interpretation function that assigns truth values to event function instances—the atoms of χ_i—, given a concrete interval ${}_j\pi_k$ over $\overline{\Psi}$. It is defined as follows:

$$I_{{}_j\pi_k}(e) \coloneqq \texttt{eval\_e}(e, {}_j\pi_k), \text{ with } e \in \mathcal{E}_i \tag{4.83}$$

Additionally, the duration constraints defined by l_i, u_i and n_i are encoded as

$$\texttt{durc}({}_j\pi_k,\, l_i,\, u_i,\, n_i) \coloneqq \texttt{duration}(|{}_j\pi_k|,\, n_1) \geq l_i \wedge \texttt{duration}(|{}_j\pi_k|,\, n_1) \leq u_i \tag{4.84}$$

Then, the set of intervals over $\overline{\Psi}$ characterized by ε_i is defined as those which satisfy both χ_i in the interpretation $I_{{}_j\pi_k}$ and the duration constraint `durc`*:*

$$\texttt{ints}(\varepsilon_i) \coloneqq \{{}_j\pi_k \mid \texttt{eval}_{I_{{}_j\pi_k}}(\chi_i) = \texttt{true} \wedge \texttt{durc}({}_j\pi_k,\, l_i,\, u_i,\, n_i) = \texttt{true}\} \tag{4.85}$$

Finally, the semantics of the events component $\varXi$ of κ is defined as the set of pairs associating event definitions with the intervals they represent as per Equation (4.85):

$$\texttt{ints}(\varXi) \coloneqq \Big\{(\varepsilon,\, \texttt{ints}(\varepsilon)) \mid \varepsilon \in \varXi\Big\} \tag{4.86}$$

As a conclusion to this subsection, Example 4.9 provides an—admittedly contrived, but illustrative—demonstration of the semantics of events, given a concrete time series.

Example 4.9 (Semantics of DTSQL Events Component) *Let $\overline{\Psi} = \langle p_1, \ldots, p_8 \rangle$ be the time series defined by Table 4.4.*

Further, let κ be a DTSQL query with events component

$$\varXi = \{\varepsilon_1, \varepsilon_2\} \tag{4.87}$$

that detects two specific kinds of events:

1. *Intervals of not more than 1.25 hours length which exhibit a monotonic increase of at least 750 %, with a tolerance against temporarily negative rates of change of 7 %.*
2. *Intervals with data points whose values are all either less than 5 or reside in the range $[6, 18]$ for at least 25 minutes.*

data point	time	identifier	value
p_1	`2022-08-21 10:00:00`	$\mathtt{t}_1$	10
p_2	`2022-08-21 10:15:00`	$\mathtt{t}_2$	2
p_3	`2022-08-21 10:30:00`	$\mathtt{t}_3$	3
p_4	`2022-08-21 10:45:00`	$\mathtt{t}_4$	5
p_5	`2022-08-21 11:00:00`	$\mathtt{t}_5$	4
p_6	`2022-08-21 11:15:00`	$\mathtt{t}_6$	6
p_7	`2022-08-21 11:30:00`	$\mathtt{t}_7$	18
p_8	`2022-08-21 11:45:00`	$\mathtt{t}_8$	19

Table 4.4: Exemplary Time Series for Example 4.9

The first event definition is formalized as

$$\begin{aligned}\varepsilon_1 &= (\mathcal{E}_1,\ \chi_1,\ l_1,\ u_1,\ n_1)\\ &= (\{e_{1,1}\},\ e_{1,1},\ 0,\ 1.25,\ \texttt{hours})\\ &= (\{\underbrace{\texttt{increase}(750,\infty,7)}_{e_{1,1}}\},\ e_{1,1},\ 0,\ 1.25,\ \texttt{hours})\end{aligned} \tag{4.88}$$

The second event definition is expressed as

$$\begin{aligned}\varepsilon_2 &= (\mathcal{E}_2,\ \chi_2,\ l_2,\ u_2,\ n_2)\\ &= (\{e_{2,1},\ e_{2,2},\ e_{2,3}\},\ (e_{2,1} \vee (\neg e_{2,2} \wedge \neg e_{2,3})),\ 25,\ \infty,\ \texttt{minutes})\\ &= (\{\underbrace{\texttt{lt}_p(5)}_{e_{2,1}},\ \underbrace{\texttt{lt}_p(6)}_{e_{2,2}},\ \underbrace{\texttt{gt}_p(18)}_{e_{2,3}}\},\ (e_{2,1} \vee (\neg e_{2,2} \wedge \neg e_{2,3})),\ 25,\ \infty,\ \texttt{minutes})\end{aligned} \tag{4.89}$$

In order to determine the value of $\texttt{ints}(\Xi)$, *first notice that* $\texttt{ints}(\varepsilon_1) = \{{}_2\pi_7\}$ *because:*

- *We have* $\texttt{chk}({}_2\pi_7) = \frac{18-2}{2} \cdot 100 = 800$ *and thus,* $\texttt{difc}({}_2\pi_7, 750, \infty) = (800 \geq 0) \wedge (800 \in [750, \infty[) = \texttt{true}$ ✓

- *As per Definition 2.8, the derivative of* Ψ *in* ${}_2\pi_7$ *with* Δx *in minutes is* $\widehat{\mathcal{P}}(2,7) = \{(\mathtt{t}_2, 0.0667), (\mathtt{t}_3, 0.1333), (\mathtt{t}_4, -0.0667), (\mathtt{t}_5, 0.1333), (\mathtt{t}_6, 0.8), (\mathtt{t}_7, 0.0667)\}$. *The only data point with a negative value component in* $\widehat{\mathcal{P}}(2,7)$ *is* $(\mathtt{t}_4, -0.0667)$. *Since* -6.67 *is* not *less than* $-t = -7$, *we have* $\texttt{ratc}({}_2\pi_7, 7) = \texttt{true}$ ✓

- *The interval cannot be extended to include* $\widehat{p_1}$ *without violating* `difc` *and* `ratc`. *It cannot be extended until* $\mathtt{t}_8$ *either, because* $\widehat{\Psi}$ *is not defined at* $\mathtt{t}_8$. *Hence, it also holds* $\texttt{maxc}({}_2\pi_7, 750, \infty, 7) = \texttt{true}$ ✓

- *We finally have* $\texttt{eval\_e}(\texttt{increase}(750, \infty, 25), {}_2\pi_7) = \texttt{true}$. *Moreover, the duration constraint* $\texttt{durc}({}_2\pi_7, 0, 1.25, \texttt{hours}) = (1.25 \geq 0) \wedge (1.25 \leq 1.25)$ *is also satisfied.* ✓

- *There is no other interval specified by ε_1 because it already covers the whole time series. Therefore, as stated, it holds that* $\texttt{ints}(\varepsilon_1) = \{{}_2\pi_7\}$.

Similarly, it can be argued that $\texttt{ints}(\varepsilon_2) = \{{}_1\pi_3,\ {}_5\pi_7\}$*:*

- *Below, Table 4.5 provides a compact overview of the value of the event filter functions employed by $\mathcal{E}_2$ across Ψ.*

time	**value**	$\texttt{lt}_p(5)$	$\texttt{lt}_p(6)$	$\texttt{gt}_p(18)$	$(\texttt{lt}_p(5) \lor (\neg\texttt{lt}_p(6) \land \neg\texttt{gt}_p(18)))$
t_1	10	`false`	`false`	`false`	`true`
t_2	2	`true`	`true`	`false`	`true`
t_3	3	`true`	`true`	`false`	`true`
t_4	5	`false`	`true`	`false`	`false`
t_5	4	`true`	`true`	`false`	`true`
t_6	6	`false`	`false`	`false`	`true`
t_7	18	`false`	`false`	`false`	`true`
t_8	19	`false`	`false`	`true`	`false`

Table 4.5: Filter Event Evaluations for Example 4.9

- *As a result of Definition 4.25 (Semantics of DTSQL Events Component), it is evident from the table that χ_2 is satisfied for the intervals ${}_1\pi_3$ and ${}_5\pi_7$.*
- *The duration constraint* $\texttt{durc}({}_i\pi_j, 20, \infty, \texttt{minutes})$*, is also satisfied for them, because* $\texttt{duration}({}_1\pi_3, \texttt{minutes}) = 30 \in [25, \infty[$ *as well as* $\texttt{duration}({}_5\pi_7, \texttt{minutes}) = 30 \in [25, \infty[$.
- *Therefore, as stated, it holds that* $\texttt{ints}(\varepsilon_2) = \{{}_1\pi_3,\ {}_5\pi_7\}$.

In conclusion, the events component of κ evaluates to

$$\texttt{ints}(\Xi) = \Big\{(\varepsilon_1,\ \{{}_2\pi_7\}),\ (\varepsilon_2,\ \{{}_1\pi_3, {}_5\pi_7\})\Big\} \tag{4.90}$$

$\triangle$

4.2.5 Selection

The selection component of a DTSQL query, if, present, gives rise to new intervals based on the intervals captured by the events component (see Section 4.2.4). To define its semantics, we first need to specify the meaning of the *precedes* and *follows* operators.

Definition 4.26 depicts the semantics of the composition operators supported by DTSQL.

Definition 4.26 (Semantics of Composition Operators) *Let Ψ be a time series, $\overline{\Psi}$ its filtered counterpart and κ a query over Ψ with events component $\varXi = \{\varepsilon_1, \dots\}$, and ω a composition operator instance. Recall that there are two different forms ω may take:*

$$op(\boxplus_1, \boxplus_2) \qquad \text{and} \qquad op(\boxplus_1, \boxplus_2, l, u, n) \tag{4.91, 4.92}$$

with $op \in \{\texttt{precedes}, \texttt{follows}\}$. This operator type *of ω may also be determined using the function* `opt` *which is defined as:*

$$\texttt{opt}(\omega) := \begin{cases} \texttt{precedes}, & \text{if } \omega = \texttt{precedes}(\boxplus_1, \boxplus_2) \text{ or} \\ & \quad \omega = \texttt{precedes}(\boxplus_1, \boxplus_2, l, u, n) \\ \texttt{follows}, & \text{if } \omega = \texttt{follows}(\boxplus_1, \boxplus_2) \text{ or} \\ & \quad \omega = \texttt{follows}(\boxplus_1, \boxplus_2, l, u, n) \end{cases} \tag{4.93}$$

Then, a composite interval constituted by ω is characterized by three criteria:

1. *One interval is represented by $\boxplus_1$, the other by $\boxplus_2$. Both $\boxplus_1$ and $\boxplus_2$ may refer to an event definition $\varepsilon \in \varXi$ or a nested operator instance $\widetilde{\omega}$.*
2. *The sequence of events represented by $\boxplus_1$ and $\boxplus_2$ must be compatible with the operator type dictated by* `opt`(ω)*.*
3. *Depending on whether a time-gap constraint is present or not, the event sequence must be immediate or within the specified time frame, respectively.*

The evaluation of a composition operator instance is denoted by the function `eval_o`(ω) *(see Equation (4.100)) which encodes the three conditions above and yields a set of intervals. In order to formalize the first constraint,* membership*, we need to introduce a function that evaluates operator arguments $\boxplus$ depending on what they represent:*

$$\texttt{eval\_a}(\boxplus) := \begin{cases} \texttt{ints}(\boxplus), & \text{if } \boxplus \in \varXi \\ \texttt{eval\_o}(\boxplus), & \text{if } \boxplus \text{ is of form (4.91) or (4.92)} \end{cases} \tag{4.94}$$

This allows the membership constraint to be expressed as

$$\texttt{memc}({}_a\pi_b, {}_c\pi_d) := {}_a\pi_b \in \texttt{eval\_a}(\boxplus_1) \land {}_c\pi_d \in \texttt{eval\_a}(\boxplus_2) \tag{4.95}$$

The second condition, specifying the temporal relation*, is equivalent to*

$$\texttt{relc}({}_a\pi_b, {}_c\pi_d, opt) := \begin{cases} \texttt{intend}({}_a\pi_b) \leq \texttt{intstart}({}_c\pi_d), & \text{if } opt = \texttt{precedes} \\ \texttt{intend}({}_c\pi_d) \leq \texttt{intstart}({}_a\pi_b), & \text{if } opt = \texttt{follows} \end{cases} \tag{4.96}$$

If no time-gap constraint is present, then the event sequence must be immediate:

$$\mathtt{immc}({}_a\pi_b, {}_c\pi_d, opt) := \begin{cases} \neg\exists p \in \mathcal{P}(\overline{\Psi})\colon \mathtt{dpt}(p) > \mathtt{intend}({}_a\pi_b) \\ \qquad\qquad \wedge\, \mathtt{dpt}(p) < \mathtt{intstart}({}_c\pi_d), & \textit{if } opt = \mathtt{precedes} \\ \neg\exists p \in \mathcal{P}(\overline{\Psi})\colon \mathtt{dpt}(p) > \mathtt{intend}({}_c\pi_d) \\ \qquad\qquad \wedge\, \mathtt{dpt}(p) < \mathtt{intstart}({}_a\pi_b), & \textit{if } opt = \mathtt{follows} \end{cases} \tag{4.97}$$

If, on the other hand, a time-gap constraint is present, then it must be satisfied:

$$\mathtt{timc}({}_a\pi_b, {}_c\pi_d, l, u, n, opt) := \mathtt{dur}({}_a\pi_b, {}_c\pi_d, n, opt) \geq l \,\wedge\, \mathtt{dur}({}_a\pi_b, {}_c\pi_d, n, opt) \leq u,$$
$$\textit{with } \mathtt{dur}({}_a\pi_b, {}_c\pi_d, n, opt) := \begin{cases} \mathtt{duration}(|{}_b\pi_c|, n), & \textit{if } opt = \mathtt{precedes} \\ \mathtt{duration}(|{}_d\pi_a|, n), & \textit{if } opt = \mathtt{follows} \end{cases} \tag{4.98}$$

These two auxiliary predicates allow the definition of the sequence *condition:*

$$\mathtt{seqc}({}_a\pi_b, {}_c\pi_d, opt) := \begin{cases} \mathtt{immc}({}_a\pi_b, {}_c\pi_d, opt), & \textit{if } \omega \textit{ is of form } (4.91) \\ \mathtt{timc}({}_a\pi_b, {}_c\pi_d, l, u, n, opt), & \textit{if } \omega \textit{ is of form } (4.92) \end{cases} \tag{4.99}$$

Finally, putting everything together, a composition operator instance ω gives rise to the following set of intervals:

$$\begin{aligned} \mathtt{eval\_o}(\omega) := \{{}_a\pi_d \mid \exists \mathsf{t_b}, \mathsf{t_c} \in \mathbb{Z}\colon & \mathtt{memc}({}_a\pi_b, {}_c\pi_d) \\ & \wedge\, \mathtt{relc}({}_a\pi_b, {}_c\pi_d, \mathtt{opt}(\omega)) \\ & \wedge\, \mathtt{seqc}({}_a\pi_b, {}_c\pi_d, \mathtt{opt}(\omega))\} \end{aligned} \tag{4.100}$$

The semantics of the selection component of a DTSQL query can be reduced to this characterization of the composite intervals represented by a composition operator, as depicted in Definition 4.27.

Definition 4.27 (Semantics of DTSQL Selection Component) *Let Ψ be a time series, $\overline{\Psi}$ its filtered counterpart and κ a query over Ψ with events component $\Xi = \{\varepsilon_1, \dots\}$, and selection component $\Omega = \omega$. Then, the set of composite intervals captured by Ω is defined as*

$$\mathtt{ints}(\Omega) := \mathtt{eval\_o}(\omega) \tag{4.101}$$

with `eval_o` *as specified in Definition 4.26.*

Ultimately, Example 4.10 provides a demonstration of how the selection component may give rise to composite events made up of atomic events following from the events component of a DTSQL query.

Example 4.10 (Semantics of DTSQL Selection Component) *Assume that κ is a query with events component $\Xi = \{\varepsilon_1, \varepsilon_2, \varepsilon_3\}$. Suppose further that the evaluation of Ξ results in periods which are illustrated schematically in Figure 4.2. The duration in minutes between two detected intervals is annotated using arrows. The shading of the sections between intervals indicates whether there are data points recorded in those sections, as explained by the figure's legend. Note also that, in practice, intervals resulting from the same event definition do not necessarily have the same length—this is only to make the illustration clearer and more easily understandable.*

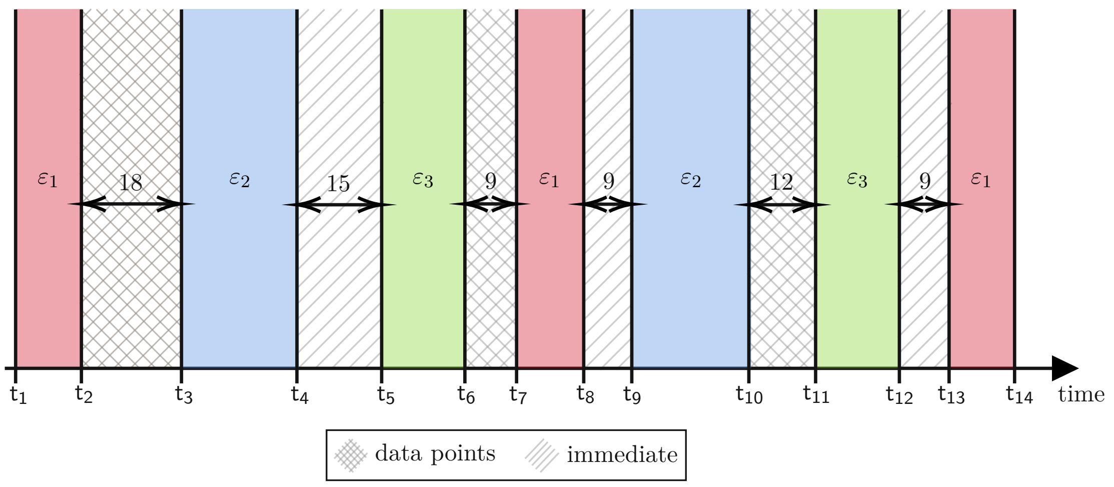


Figure 4.2: Detected Events for Example 4.10

Now assume κ has the following selection component:

$$\Omega = \underbrace{\texttt{follows}(\varepsilon_1,\ \overbrace{\texttt{precedes}(\varepsilon_2,\ \varepsilon_3,\ 0,\ 15,\ \texttt{minutes})}^{\omega_1})}_{\omega_2} \tag{4.102}$$

The (recursive) evaluation of Ω yields:

$$\begin{aligned} \texttt{eval\_o}(\omega_1) &= \{{}_3\pi_6,\ {}_9\pi_{12}\} \\ \texttt{eval\_o}(\omega_2) = \texttt{eval\_o}(\Omega) &= \{{}_9\pi_{14}\} \end{aligned} \tag{4.103}$$

Regarding ω_1, both ${}_3\pi_6$ and ${}_9\pi_{12}$ evidently satisfy memc*,* relc*,* seqc*. In the case of ω_2, there are two possible intervals—${}_3\pi_8$ and ${}_9\pi_{14}$. The former one, ${}_3\pi_8$ does* not *satisfy* secq *because the time frame from* t₆ *to* t₈ *contains data points. This is invalid since, due to ω_1 not exhibiting a time-gap constraint, ${}_7\pi_8$ should* immediately *follow ${}_3\pi_6$. The latter one, ${}_9\pi_{14}$, satisfies this constraint because there are no data points between* t₁₂ *and* t₁₃*. It also satisfies the other two constraints—*memc *and* relc*—, making the composite (merged) interval ${}_9\pi_{14}$ the result of evaluating $\omega_2 = \Omega$.* △

4.2.6 Yield

The ultimate result of a DTSQL query is determined by the yield component. Its concrete semantics are specified in Definition 4.28.

Definition 4.28 (Semantics of DTSQL Yield Component) *Let Ψ be a time series and $\kappa = (\mathcal{S}, \Phi, \Xi, \Omega, \Upsilon)$ a query with yield component $\Upsilon = (\gamma, d)$. Then, the result of κ given Ψ, denoted $\texttt{res}(\kappa, \Psi)$, is defined via a case distinction on γ.*

Case 1: $\gamma = \texttt{allints}$
If neither events nor selection component is present, an empty set of intervals is returned. If only the events component is present, then the set of captured event intervals is returned. If both events and selection component are present, all detected composite intervals are returned. More concretely:

$$\texttt{res}(\kappa, \Psi) := \begin{cases} \emptyset, & \textit{if } \Xi = \emptyset \land \Omega = \square \\ \bigcup\limits_{\varepsilon \in \Xi} \texttt{ints}(\varepsilon), & \textit{if } \Xi \neq \emptyset \land \Omega = \square \\ \texttt{ints}(\Omega), & \textit{if } \Xi \neq \emptyset \land \Omega \neq \square \end{cases} \tag{4.104}$$

The fourth case ($\Xi = \emptyset \land \Omega \neq \square$) is syntactically invalid and therefore undefined.

Case 2: $\gamma = \texttt{maxints}$
The set of detected intervals with maximum length is returned. Let Π be the set of all detected intervals according to Equation (4.104). Then:

$$\texttt{res}(\kappa, \Psi) := \{{}_i\pi_j \in \Pi \mid |{}_i\pi_j| = \max_{{}_a\pi_b \in \Pi} |{}_a\pi_b|\} \tag{4.105}$$

Case 3: $\gamma = \texttt{minints}$
The set of detected intervals with minimum length is returned. Let Π be the set of all detected intervals according to Equation (4.104). Then:

$$\texttt{res}(\kappa, \Psi) := \{{}_i\pi_j \in \Pi \mid |{}_i\pi_j| = \min_{{}_a\pi_b \in \Pi} |{}_a\pi_b|\} \tag{4.106}$$

Case 4: $\gamma = \texttt{datapoints}$
All data points that have not been filtered out and/or are contained in at least one detected interval are returned. Let Π bet the set of all detected intervals according to Equation (4.104). Then:

$$\texttt{res}(\kappa, \Psi) := \begin{cases} \{p \in \mathcal{P}(\overline{\Psi})\}, & \textit{if } \Xi = \emptyset \\ \bigcup\limits_{{}_i\pi_j \in \Pi} \texttt{ps}_{\overline{\Psi}}({}_i\pi_j), & \textit{if } \Xi \neq \emptyset \end{cases} \tag{4.107}$$

Case 5: $\gamma = \texttt{sample}$, $d = s$
The computed value of the sample s is returned:

$$\texttt{res}(\kappa, \Psi) \coloneqq \texttt{eval\_s}(s) \tag{4.108}$$

Case 6: $\gamma = \texttt{samples}$, $d = S$
The computed values of the samples represented by S are returned:

$$\texttt{res}(\kappa, \Psi) \coloneqq \{\texttt{eval\_s}(s) \mid s \in S\} \tag{4.109}$$

The final Example 4.11 illustrates the semantics of different yield formats given a set of detected intervals.

Example 4.11 *Assume Ψ is a time series and $\kappa = (\mathcal{S}, \Phi, \Xi, \Omega, \Upsilon)$ a DTSQL query. Assume further that the set of computed samples is equal to $\mathcal{S}^c = \{\underbrace{195.12}_{s_1}, \underbrace{24.39}_{s_2}, \underbrace{31.81}_{s_3}\}$ and the sets of intervals detected as a result of Ξ, Ω and Equation (4.104) is $\Pi = \{{}_3\pi_5, {}_{15}\pi_{16}, {}_{19}\pi_{21}\}$ with each one containing the data points*

$$\begin{aligned}
\texttt{ps}_{\overline{\Psi}}({}_3\pi_5) &= \{(\mathsf{t}_3, 17), (\mathsf{t}_4, 25.3), (\mathsf{t}_5, 24.25)\} \\
\texttt{ps}_{\overline{\Psi}}({}_{15}\pi_{16}) &= \{(\mathsf{t}_{15}, 24), (\mathsf{t}_{16}, 31.81)\} \\
\texttt{ps}_{\overline{\Psi}}({}_{19}\pi_{21}) &= \{(\mathsf{t}_{19}, 23.63), (\mathsf{t}_{20}, 28.01), (\mathsf{t}_{21}, 21.12)\}
\end{aligned} \tag{4.110}$$

Then, possible yield components along with their query results include:

$$\begin{aligned}
\Upsilon_1 &= (\texttt{allints}, \square), & \texttt{res}(\kappa, \Psi)_1 &= \{{}_3\pi_5, {}_{15}\pi_{16}, {}_{19}\pi_{21}\} \\
\Upsilon_2 &= (\texttt{maxints}, \square), & \texttt{res}(\kappa, \Psi)_2 &= \{{}_3\pi_5, {}_{19}\pi_{21}\} \\
\Upsilon_3 &= (\texttt{minints}, \square), & \texttt{res}(\kappa, \Psi)_3 &= \{{}_{15}\pi_{16}\} \\
\Upsilon_4 &= (\texttt{datapoints}, \square), & \texttt{res}(\kappa, \Psi)_4 &= \texttt{ps}_{\overline{\Psi}}({}_3\pi_5) \cup \texttt{ps}_{\overline{\Psi}}({}_{15}\pi_{16}) \cup \texttt{ps}_{\overline{\Psi}}({}_{19}\pi_{21}) \\
\Upsilon_5 &= (\texttt{sample}, s_3), & \texttt{res}(\kappa, \Psi)_5 &= 31.81 \\
\Upsilon_6 &= (\texttt{samples}, \{s_1, s_2\}), & \texttt{res}(\kappa, \Psi)_6 &= \{192.12, 24.39\}
\end{aligned} \tag{4.111}$$

△

4.3 Language Grammar

The preceding Section 4.1 has introduced the abstract syntax of DTSQL. In this section, we present a language grammar that corresponds to that syntax specification. Note that there are (many) other grammars which could still be a valid representation of the syntax described in Section 4.1. The specific manifestation depicted in this section forms the basis of the reference implementation detailed in Chapter 5.

The grammar is specified in ANTLR4[1] syntax. This bridges the mental gap between the language specification and its implementation because ANTLR will be used in Chapter 5 to generate a lexer and parser to process DTSQL queries. The syntax of ANTLR grammars is very similar to EBNF (extended Backus-Naur form), which makes it relatively intuitive.

According to ANTLR's notational conventions, the names of parser rules start with a lowercase letter and names of lexer rules start with an uppercase letter. Furthermore, the definition of auxiliary parser or lexer rules which are referenced by multiple other rules are mentioned only at their first appearance.

The following subsections each depict the ANTLR definition of the respective DTSQL query component, including an explanation of the rules involved. Moreover, examples of these components corresponding to the abstract examples in Section 4.1 will be provided in the concrete syntax.

On a final general note, in the grammar presented below, whitespaces generally do not bear any meaning. There are isolated instances where at least one whitespace is mandatory, but apart from that, the formatting of a DTSQL query is arbitrary and does not influence its validity or semantics.

4.3.1 DTSQL Query

The grammar of valid DTSQL queries is the result of combining the grammars of all individual query components, as presented in Listing 4.1. As lines 2–7 show, the components are required to follow each other in the order they have been introduced in Section 4.1 and Section 4.2.

```
1  dtsqlQuery
2    :  WHITESPACE?
3         (samplesDeclaration WHITESPACE)?
4         (filtersDeclaration WHITESPACE)?
5         (eventsDeclaration WHITESPACE)?
6         (selectDeclaration WHITESPACE)?
7         yieldDeclaration WHITESPACE?
8         EOF  ;
9
10 WHITESPACE   :  WHITESPACE_CHARACTER+   ;
11 fragment WHITESPACE_CHARACTER
12   :  ' '   |  '\r'   |  '\n'   |  '\r\n'   |  '\t'   ;
```

Listing 4.1: ANTLR Grammar for a DTSQL Query

The parser rules referenced in the listing above are presented in the following subsections and therefore, to avoid unnecessary repetitions, not elaborated on at this point. However, for the sake of completeness, the full, uncompressed grammar is provided in Appendix A.1. Parser rules are listed in Appendix A.1.1 and lexer rules in Appendix A.1.2.

The initial Example 4.12 gives an overview of the core features of the concrete DTSQL grammar. Detailed explanations on the individual parts follow in the next subsections.

[1] https://www.antlr.org/

Example 4.12 (Concrete Syntax of DTSQL Query) *Consider the query from Example 4.6 (Semantics of DTSQL Query), i.e.,*

$$\begin{aligned}\kappa = \Big(&\{\mathtt{avg}_\Psi\},\\ &(\underbrace{\{\mathtt{around\_rel}_p(\mathtt{avg}_\Psi, 25)\}}_{f_1}, \neg f_1),\\ &\{\underbrace{(\overbrace{\{\mathtt{lt}_p(\mathtt{avg}_\Psi)\}}^{e_{1,1}}, e_{1,1}, 0, \infty, \mathtt{minutes})}_{\varepsilon_1}, \underbrace{(\overbrace{\{\mathtt{gt}_p(\mathtt{avg}_\Psi)\}}^{e_{2,1}}, e_{2,1}, 0, \infty, \mathtt{minutes})}_{\varepsilon_2}\},\\ &\mathtt{precedes}(\varepsilon_1, \varepsilon_2),\\ &(\mathtt{datapoints}, \square)\Big)\end{aligned} \tag{4.112}$$

The concrete plain-text representation of κ is presented in Listing 4.2. Observe that the grammar allows assigning human-readable identifiers to samples and event definitions, which facilitates referencing them in other query components.

```
WITH SAMPLES:
  avg() AS globalAverage
APPLY FILTER:
  AND(NOT(around(rel, globalAverage, 25)))
USING EVENTS:
  OR(lt(globalAverage)) AS low,
  AND(gt(globalAverage)) AS high
SELECT PERIODS:
  (low precedes high)
YIELD:
  data points
```

Listing 4.2: Concrete Syntax of a Basic DTSQL Query

△

4.3.2 Samples

The concrete grammar of the DTSQL samples component is depicted in Listing 4.3. Afterwards, Example 4.13 presents a samples component in this syntax.

```
samplesDeclaration  :  SAMPLES_CLAUSE COLON WHITESPACE
    ↪ aggregatorsDeclarationStatement  ;
aggregatorsDeclarationStatement  :  aggregatorList  ;

aggregatorList
  :  aggregators LIST_SEPARATOR aggregatorDeclaration
  |  aggregatorDeclaration  ;
aggregators : aggregatorDeclaration (LIST_SEPARATOR aggregatorDeclaration)* ;

```

```
 9 aggregatorDeclaration  :  aggregatorFunctionDeclaration WHITESPACE
    ↪ identifierDeclaration  ;
10 aggregatorFunctionDeclaration  :  valueAggregatorDeclaration  |
    ↪ temporalAggregatorDeclaration  ;
11
12 valueAggregatorDeclaration  :  VALUE_AGGREGATOR_FUNCTION PARENTHESIS_OPEN
    ↪ WHITESPACE? timeRange? WHITESPACE? PARENTHESIS_CLOSE  ;
13 temporalAggregatorDeclaration
14   :  TEMPORAL_AGGREGATOR_FUNCTION PARENTHESIS_OPEN WHITESPACE? TIME_UNIT
    ↪ LIST_SEPARATOR intervalList WHITESPACE? PARENTHESIS_CLOSE
15   |  UNITLESS_TEMPORAL_AGGREGATOR_FUNCTION PARENTHESIS_OPEN WHITESPACE?
    ↪ intervalList WHITESPACE? PARENTHESIS_CLOSE  ;
16
17 intervalList
18   :  intervals LIST_SEPARATOR STRING_LITERAL
19   |  STRING_LITERAL  ;
20 intervals  :  STRING_LITERAL (LIST_SEPARATOR STRING_LITERAL)*  ;
21
22 timeRange  :  STRING_LITERAL LIST_SEPARATOR STRING_LITERAL  ;
23 identifierDeclaration  :  AS WHITESPACE IDENTIFIER  ;
24
25 SAMPLES_CLAUSE  :  'WITH SAMPLES'  ;
26
27 VALUE_AGGREGATOR_FUNCTION  :  'avg'  |  'max'  |  'min'  |  'sum'  |  'count'
    ↪  |  'integral'  |  'stddev'  ;
28 TEMPORAL_AGGREGATOR_FUNCTION  :  'avg_t'  |  'max_t'  |  'min_t'  |  'sum_t'
    ↪  |  'stddev_t'  ;
29 UNITLESS_TEMPORAL_AGGREGATOR_FUNCTION  :  'count_t'  ;
30 TIME_UNIT  :  'weeks'  |  'days'  |  'hours'  |  'minutes'  |  'seconds'  |
    ↪ 'millis'  ;
31
32 PARENTHESIS_OPEN  :  '('  ;
33 PARENTHESIS_CLOSE  :  ')'  ;
34 COLON  :  ':'  ;
35 fragment COMMA  :  ','  ;
36 LIST_SEPARATOR  :  WHITESPACE? COMMA WHITESPACE?  ;
37
38 IDENTIFIER  :  IDENTIFIER_FIRST_CHARACTER IDENTIFIER_CHARACTER*  ;
39 fragment IDENTIFIER_FIRST_CHARACTER  :  LETTER_CHARACTER  ;
40 fragment IDENTIFIER_CHARACTER  :  LETTER_CHARACTER  |  DIGIT_CHARACTER  ;
41
42 STRING_LITERAL  :  '"' STRING_CHARACTERS? '"'  ;
43 fragment STRING_CHARACTERS  :  STRING_CHARACTER+  ;
44 fragment STRING_CHARACTER  :  ~["\\\r\n]  ;
```

Listing 4.3: ANTLR Grammar for the Samples Component in a DTSQL Query

Explanation

- Lines 1–2: The samples component starts with `WITH SAMPLES:␣`, followed by an aggregator list.

- Lines 4–7: The aggregator list consists of either two or more aggregators (line 5), separated by `,`, or exactly one (line 6).
- Lines 9–10: An aggregator is defined by an aggregator function, followed by an identifier. Aggregator functions either operate on the value or the temporal dimension.
- Lines 12–15: Value aggregators are local if their `timeRange` component consisting of two date literals is present, otherwise global. The specific format of the date literals is implementation-specific. Temporal aggregators expect a time unit (except for `count_t`) determining the scale of its value and a list of interval literals. The specific format for specifying an interval is also up to the implementation.

Example 4.13 (Concrete Syntax of DTSQL Samples) *Consider the samples component specified in Example 4.1 (DTSQL Samples Component), i.e.,*

$$\mathcal{S} = \{\texttt{max}_\Psi,\ \texttt{integral}_\Psi(\texttt{t}_1, \texttt{t}_4),\ \texttt{sum\_t}_\Psi(\texttt{minutes}, \{{}_3\pi_8,\ {}_{10}\pi_{12},\ {}_{15}\pi_{23}\})\} \tag{4.113}$$

Listing 4.4 expresses $\mathcal{S}$ in the concrete grammar, where time identifiers $\mathsf{t_i}$ represent the point in time `2022-08-28 14:00:00` plus i hours.

```
WITH SAMPLES:
  max() AS globalAverage,
  integral("2022-08-28T15:00:00Z", "2022-08-28 16:00:00Z") AS localIntegral,
  sum_t(minutes, "2022-08-28T17:00:00Z/2022-08-28T22:00:00Z",
                 "2022-08-29T00:00:00Z/2022-08-29T02:00:00Z",
                 "2022-08-29T05:00:00Z/2022-08-29T13:00:00Z") AS temporalSum
```

Listing 4.4: Concrete Syntax of a DTSQL Samples Component

△

4.3.3 Filters

Listing 4.5 provides an overview of the concrete grammar of the DTSQL filter component. Moreover, an exemplary filter component expressed in this grammar is depicted Example 4.14.

```
1 filtersDeclaration   :  FILTER_CLAUSE COLON WHITESPACE filterConnective  ;
2 filterConnective   :  CONNECTIVE_IDENTIFIER PARENTHESIS_OPEN WHITESPACE?
    ↪ singlePointFilterList WHITESPACE? PARENTHESIS_CLOSE  ;
3
4 singlePointFilterList
5   :  singlePointFilters LIST_SEPARATOR singlePointFilterDeclaration
6   |  singlePointFilterDeclaration  ;
7 singlePointFilters  :  singlePointFilterDeclaration (LIST_SEPARATOR
    ↪ singlePointFilterDeclaration)*  ;
```

```
 8
 9 singlePointFilterDeclaration  :  singlePointFilter  |
       ↪ negatedSinglePointFilter  ;
10 singlePointFilter : thresholdFilter  |  temporalFilter  |  deviationFilter ;
11 negatedSinglePointFilter  :  CONNECTIVE_NOT PARENTHESIS_OPEN WHITESPACE?
       ↪ singlePointFilter WHITESPACE? PARENTHESIS_CLOSE  ;
12
13 temporalFilter  :  TEMPORAL_FILTER_TYPE PARENTHESIS_OPEN WHITESPACE?
       ↪ STRING_LITERAL WHITESPACE? PARENTHESIS_CLOSE  ;
14 thresholdFilter  :  THRESHOLD_FILTER_TYPE PARENTHESIS_OPEN WHITESPACE?
       ↪ scalarArgument WHITESPACE? PARENTHESIS_CLOSE  ;
15 deviationFilter  :  DEVIATION_FILTER_TYPE PARENTHESIS_OPEN WHITESPACE?
       ↪ deviationFilterArguments WHITESPACE? PARENTHESIS_CLOSE  ;
16
17 scalarArgument  :  NUMBER  |  IDENTIFIER  ;
18 deviationFilterArguments  :  AROUND_FILTER_TYPE LIST_SEPARATOR reference=
       ↪ scalarArgument LIST_SEPARATOR deviation=scalarArgument  ;
19
20 FILTER_CLAUSE  :  'APPLY FILTER'  ;
21
22 CONNECTIVE_NOT  :  'NOT'  ;
23 CONNECTIVE_IDENTIFIER  :  'AND'  |  'OR'  ;
24
25 THRESHOLD_FILTER_TYPE  :  'gt'  |  'lt'  ;
26 TEMPORAL_FILTER_TYPE  :  'before'  |  'after'  ;
27 DEVIATION_FILTER_TYPE  :  'around'  ;
28 AROUND_FILTER_TYPE  :  'rel'  |  'abs'  ;
29
30 NUMBER  :  INT  |  FLOAT  ;
31 fragment DIGIT  :  [0-9]  ;
32 fragment SIGN  :  '-'?  ;
33 INT  :  SIGN? DIGIT+  ;
34 FLOAT  :  SIGN? DIGIT+ '.' DIGIT+  ;
```

Listing 4.5: ANTLR Grammar for the Filter Component in a DTSQL Query

Explanation

- Lines 1–2: The filter component starts with `APPLY FILTER:␣`, followed by a filter connective consisting of a root connective `AND` or `OR` and a list of filters as arguments.
- Lines 4–7: A (single point) filter list is made up of either two or more filters (line 5), separated by `,`, or exactly one (line 6).
- Lines 9–11: Filters may be negated and take the form of a threshold, temporal or deviation filter.
- Lines 13–15: Temporal filters accept a date literal as argument, threshold filters one scalar (which may reference a sample), and deviation filters a type (relative or absolute) as well as two scalars.

Example 4.14 (Concrete Syntax of DTSQL Filters) *Consider the filter component specified in Example 4.2 (DTSQL Filter Component), i.e.,*

$$\Phi = (\{\underbrace{\texttt{after}_p(\texttt{t}_{11})}_{f_1}, \underbrace{\texttt{around\_rel}_p(\texttt{avg}_\Psi, 5.25)}_{f_2}\}, (f_1 \wedge \neg f_2)) \tag{4.114}$$

The concrete syntax of Φ *is depicted in Listing 4.6, where* $\texttt{t}_{11}$ *represents the point in time 2022-08-28 14:00:00 and the sample* $\texttt{avg}_\Psi$ *is denoted by the identifier avg1.*

```
WITH SAMPLES:
  avg() AS avg1
APPLY FILTER:
  AND(
      after("2022-08-28T14:00:00.000Z"),
      NOT(around(rel, avg1, 5.25))
     )
```

Listing 4.6: Concrete Syntax of a DTSQL Filter Component

△

4.3.4 Events

In Listing 4.7, the concrete grammar of the DTSQL events component is shown. In addition to that, Example 4.15 demonstrates a specific events component in this syntax.

```
eventsDeclaration  :  EVENTS_CLAUSE COLON WHITESPACE
    ↪ eventsDeclarationStatement  ;
eventsDeclarationStatement  :  eventList  ;

eventList
  :  events LIST_SEPARATOR eventDeclaration
  |  eventDeclaration  ;
events  :  eventDeclaration (LIST_SEPARATOR eventDeclaration)*  ;

eventDeclaration  :  eventConnective WHITESPACE? (durationSpecification
    ↪ WHITESPACE?)? identifierDeclaration  ;
durationSpecification  : EVENT_DURATION WHITESPACE TIME_UNIT  ;
eventConnective  :  CONNECTIVE_IDENTIFIER PARENTHESIS_OPEN WHITESPACE?
    ↪ eventFunctionList WHITESPACE? PARENTHESIS_CLOSE  ;

eventFunctionList
  :  eventFunctions LIST_SEPARATOR eventFunctionDeclaration
  |  eventFunctionDeclaration  ;
eventFunctions  :  eventFunctionDeclaration (LIST_SEPARATOR
    ↪ eventFunctionDeclaration)*  ;
eventFunctionDeclaration  :  singlePointFilterDeclaration  |
    ↪ complexEventDeclaration  ;

complexEventDeclaration  :  complexEvent  |  negatedComplexEvent  ;
complexEvent  :  constantEvent  |  increaseEvent  |  decreaseEvent  ;
```

```
21 negatedComplexEvent  :  CONNECTIVE_NOT PARENTHESIS_OPEN WHITESPACE?
    ↪ complexEvent WHITESPACE? PARENTHESIS_CLOSE  ;
22
23 constantEvent  :  EVENT_CONSTANT PARENTHESIS_OPEN WHITESPACE? slope=
    ↪ scalarArgument LIST_SEPARATOR deviation=scalarArgument WHITESPACE?
    ↪ PARENTHESIS_CLOSE  ;
24 increaseEvent  :  EVENT_INCREASE PARENTHESIS_OPEN WHITESPACE? minChange=
    ↪ scalarArgument LIST_SEPARATOR monotonicUpperBound LIST_SEPARATOR
    ↪ tolerance=scalarArgument WHITESPACE? PARENTHESIS_CLOSE  ;
25 decreaseEvent  :  EVENT_DECREASE PARENTHESIS_OPEN WHITESPACE? minChange=
    ↪ scalarArgument LIST_SEPARATOR monotonicUpperBound LIST_SEPARATOR
    ↪ tolerance=scalarArgument WHITESPACE? PARENTHESIS_CLOSE  ;
26
27 monotonicUpperBound  :  scalarArgument  |  HYPHEN  ;
28
29 EVENTS_CLAUSE  :  'USING EVENTS'  ;
30 AS  :  'AS'  ;
31 EVENT_CONSTANT  :  'const'  ;
32 EVENT_INCREASE  :  'increase'  ;
33 EVENT_DECREASE  :  'decrease'  ;
34 HYPHEN  :  '-'  ;
35
36 EVENT_DURATION  :  DURATION_FOR WHITESPACE DURATION_RANGE_OPEN WHITESPACE?
    ↪ INT? LIST_SEPARATOR INT? DURATION_RANGE_CLOSE  ;
37 fragment DURATION_FOR  :  'FOR'  ;
38 fragment DURATION_RANGE_OPEN  :  PARENTHESIS_OPEN  |  '['  ;
39 fragment DURATION_RANGE_CLOSE  :  PARENTHESIS_CLOSE  | ']'  ;
```

Listing 4.7: ANTLR Grammar for the Events Component in a DTSQL Query

Explanation

- Lines 3–7: The events component starts with `USING EVENTS:␣`, followed by an event list.
- Lines 4–7: An event list consists of either two or more event declarations (line 5), separated by `,`, or exactly one (line 6).
- Lines 9–11: An event declaration is mainly constituted by an event connective, but also has an identifier and—optionally—a minimum and/or maximum duration.
- Lines 13–17: Event connectives are represented by a list of one (line 15) or more (line 14) event function declarations, separated by `,`. An event function declaration either represents a filter event, or a complex event.
- Lines 19–21: Complex events may be negated and correspond to constant, increase or decrease events.
- Lines 23–27: While constant events accept two scalar arguments, increase and decrease events receive three scalar arguments. For monotonic upper bounds, a hyphen (-) is also a valid value, representing infinity ("no upper bound").

Example 4.15 (Concrete Syntax of DTSQL Events) *Consider the events component specified in Example 4.3 (DTSQL Events Component), i.e., $\Xi = \{\varepsilon_1, \varepsilon_2\}$ with*

$$\begin{aligned}\varepsilon_1 &= (\{\underbrace{\mathtt{gt}_p(300)}_{e_{1,1}}, \underbrace{\mathtt{lt}_p(100)}_{e_{1,2}}\}, (e_{1,1} \lor e_{1,2}), 30, \infty, \mathtt{minutes}) \\ \varepsilon_2 &= (\{\underbrace{\mathtt{increase}(50, \infty, 5.75)}_{e_{2,1}}, \underbrace{\mathtt{before}_p(\mathtt{t}_{23})}_{e_{2,2}}\}, (e_{2,1} \land \neg e_{2,2}), 0, 45, \mathtt{seconds})\end{aligned} \tag{4.115}$$

The concrete syntax of Φ is depicted in Listing 4.8, where $\mathtt{t}_{23}$ represents the point in time `2022-08-28 17:45:23`.

```
USING EVENTS:
  OR(gt(300), lt(100)) FOR [30,] minutes AS event1,
  AND(increase(50, -, 5.75),
      NOT(before("2022-08-28 17:45:23.000Z"))
     ) FOR [0,45] seconds AS event2
```

Listing 4.8: Concrete Syntax of a DTSQL Events Component

△

4.3.5 Selection

Listing 4.9 presents the concrete grammar of the DTSQL selection component. After that, a specific selection component according to this syntax is given in Example 4.16.

```
selectDeclaration  :  SELECT_CLAUSE COLON WHITESPACE temporalRelation  ;

temporalRelation
  :  PARENTHESIS_OPEN op1=IDENTIFIER WHITESPACE TEMPORAL_RELATION WHITESPACE
   ↪ op2=IDENTIFIER WHITESPACE? timeToleranceSpecification?
   ↪ PARENTHESIS_CLOSE  #EventEvent
  |  PARENTHESIS_OPEN op1=IDENTIFIER WHITESPACE TEMPORAL_RELATION WHITESPACE
   ↪ op2=temporalRelation WHITESPACE? timeToleranceSpecification?
   ↪ PARENTHESIS_CLOSE  #EventRecursive
  |  PARENTHESIS_OPEN op1=temporalRelation WHITESPACE TEMPORAL_RELATION
   ↪ WHITESPACE op2=IDENTIFIER WHITESPACE? timeToleranceSpecification?
   ↪ PARENTHESIS_CLOSE  #RecursiveEvent
  |  PARENTHESIS_OPEN op1=temporalRelation WHITESPACE TEMPORAL_RELATION
   ↪ WHITESPACE op2=temporalRelation WHITESPACE? timeToleranceSpecification
   ↪ ? PARENTHESIS_CLOSE  #RecursiveRecursive  ;

timeToleranceSpecification  : TIME_TOLERANCE WHITESPACE TIME_UNIT  ;

SELECT_CLAUSE  :  'SELECT PERIODS'  ;
TEMPORAL_RELATION  :  'precedes'  |  'follows'  ;

TIME_TOLERANCE  :  TIME_TOLERANCE_WITHIN WHITESPACE DURATION_RANGE_OPEN
   ↪ WHITESPACE? INT? LIST_SEPARATOR INT? DURATION_RANGE_CLOSE  ;
```

```
fragment TIME_TOLERANCE_WITHIN   :  'WITHIN'  ;
fragment TIME_TOLERANCE_OPEN   :  PARENTHESIS_OPEN   |  '['   ;
fragment TIME_TOLERANCE_CLOSE   :  PARENTHESIS_CLOSE   | ']'   ;
```

Listing 4.9: ANTLR Grammar for the Selection Component in a DTSQL Query

Explanation

- Line 1: The selection component starts with `SELECT PERIODS:␣`, followed by a (potentially nested) temporal relation between (composite) events.
- Lines 3–7: Valid temporal relation operators are `precedes` and `follows`. The temporal relations supported by DTSQL are binary and arguments may represent either an event (via its identifier) or a recursive operator instance. Therefore, the rule `temporalRelation` covers all four permutations—with each being labelled accordingly, e.g., `#EventEvent` in line 5.

Example 4.16 (Concrete Syntax of DTSQL Selection) *Consider the selection component Ω_4 specified in Example 4.4 (DTSQL Selection Component), i.e.,*

$$\Omega_4 = \texttt{precedes}(\varepsilon_2, \texttt{follows}(\varepsilon_4, \varepsilon_1), 0, 30, \texttt{minutes}) \tag{4.116}$$

A representation of Ω_4 in its concrete grammar is depicted in Listing 4.10, where the event definitions ε_1, ε_2 and ε_4 are identified by `event1`, `event2` and `event4`, respectively.

```
USING EVENTS:
  (...) AS event1,
  (...) AS event2,
  (...) AS event4
SELECT PERIODS:
  (
    event2
    precedes
    (event4 follows event1)
  ) WITHIN [0,30] minutes
```

Listing 4.10: Concrete Syntax of a DTSQL Selection Component

△

4.3.6 Yield

The grammar of the DTSQL yield component is specified in Listing 4.11. Subsequently, Example 4.17 depicts a specific yield component compatible with this grammar.

```
yieldDeclaration   :  YIELD COLON WHITESPACE yieldType   ;
yieldType
  :  YIELD_ALL_PERIODS
```

```
 4   |  YIELD_LONGEST_PERIOD
 5   |  YIELD_SHORTEST_PERIOD
 6   |  YIELD_DATA_POINTS
 7   |  YIELD_SAMPLE WHITESPACE IDENTIFIER
 8   |  YIELD_SAMPLE_SET WHITESPACE identifierList  ;
 9
10 identifierList
11   :  identifiers LIST_SEPARATOR IDENTIFIER
12   |  IDENTIFIER  ;
13 identifiers  :  IDENTIFIER (LIST_SEPARATOR IDENTIFIER)*  ;
14
15 YIELD  :  'YIELD'  ;
16 YIELD_ALL_PERIODS  :  'all periods'  ;
17 YIELD_LONGEST_PERIOD  :  'longest period'  ;
18 YIELD_SHORTEST_PERIOD  :  'shortest period'  ;
19 YIELD_DATA_POINTS  :  'data points'  ;
20 YIELD_SAMPLE  :  'sample'  ;
21 YIELD_SAMPLE_SET :  'samples'  ;
```

Listing 4.11: ANTLR Grammar for the Yield Component in a DTSQL Query

Explanation

- Line 1: The yield component starts with `YIELD:␣`, followed by a yield type.
- Line 2: The majority of yield types do not require arguments. However, `sample` expects one sample identifier and `samples` a list of them.

Example 4.17 (Concrete Syntax of DTSQL Yield) *Consider the yield components specified in Example 4.5 (DTSQL Yield Component), i.e.,*

$$\begin{aligned}
\Upsilon_1 &= (\texttt{allints}, \Box)\\
\Upsilon_2 &= (\texttt{maxints}, \Box)\\
\Upsilon_3 &= (\texttt{minints}, \Box)\\
\Upsilon_4 &= (\texttt{datapoints}, \Box)\\
\Upsilon_5 &= (\texttt{sample}, s_2)\\
\Upsilon_6 &= (\texttt{samples}, \{s_1, s_3, s_4\})
\end{aligned} \tag{4.117}$$

Listing 4.12 depicts concrete representations of Υ_1 through Υ_6, where the samples s_i are identified as `samplei`, respectively, with $i \in \{1, \ldots, 4\}$.

```
YIELD: all periods
YIELD: longest period
YIELD: shortest period
YIELD: data points
YIELD: sample sample2
YIELD: samples sample1, sample3, sample4
```

Listing 4.12: Concrete Syntax of a DTSQL Yield Component

△

CHAPTER 5

Reference Implementation

This chapter goes into detail about how the reference implementation of the time series query language DTSQL specified in Chapter 4 has been created. First, a general overview of the system architecture is presented. This is followed by a section dedicated to the process of parsing and evaluating incoming queries. The chapter concludes with a description of the client environment which assists users in conveniently creating and executing DTSQL queries over their time series data.

5.1 Architecture Overview

The DTSQL system proposed as part of this thesis is designed with extensibility in mind. Its query routines operate on a simple, *canonical* representation of time series which essentially corresponds to how they were defined in Section 2.4.2, i.e., a list of time-value pairs. In order to support arbitrary storage solutions (e.g., an InfluxDB database, a local CSV file, data accessible via a web service, ...), there is a `StorageService` interface which defines the following main routines:

- `load`: Reads a time series from the respective storage solution and loads it in a storage-specific format into memory.
- `transform`: Defines a transformation from the storage-specific format into the canonical representation of a time series used by the system.
- `store`: Persists a time series in canonical representation in the respective storage solution.

If there is demand to add DTSQL query support for a new storage solution—one might want to query time series in a specific SQL database—one merely needs to implement these methods defined by the `StorageService` interface. They accept storage-specific configurations as parameters. A CSV implementation, for instance, requires information about which column in a file denotes the time of a data point, or which format should be used to parse the date and time values.

The diagram depicted in Figure 5.1 shows a high-level overview of the most important components that constitute the DTSQL reference implementation. It also emphasizes the sequence of steps conducted in the process of creating, interpreting and evaluating DTSQL queries. The enumeration below gives a more in-depth explanation of the system architecture by elaborating on each step highlighted in the figure.

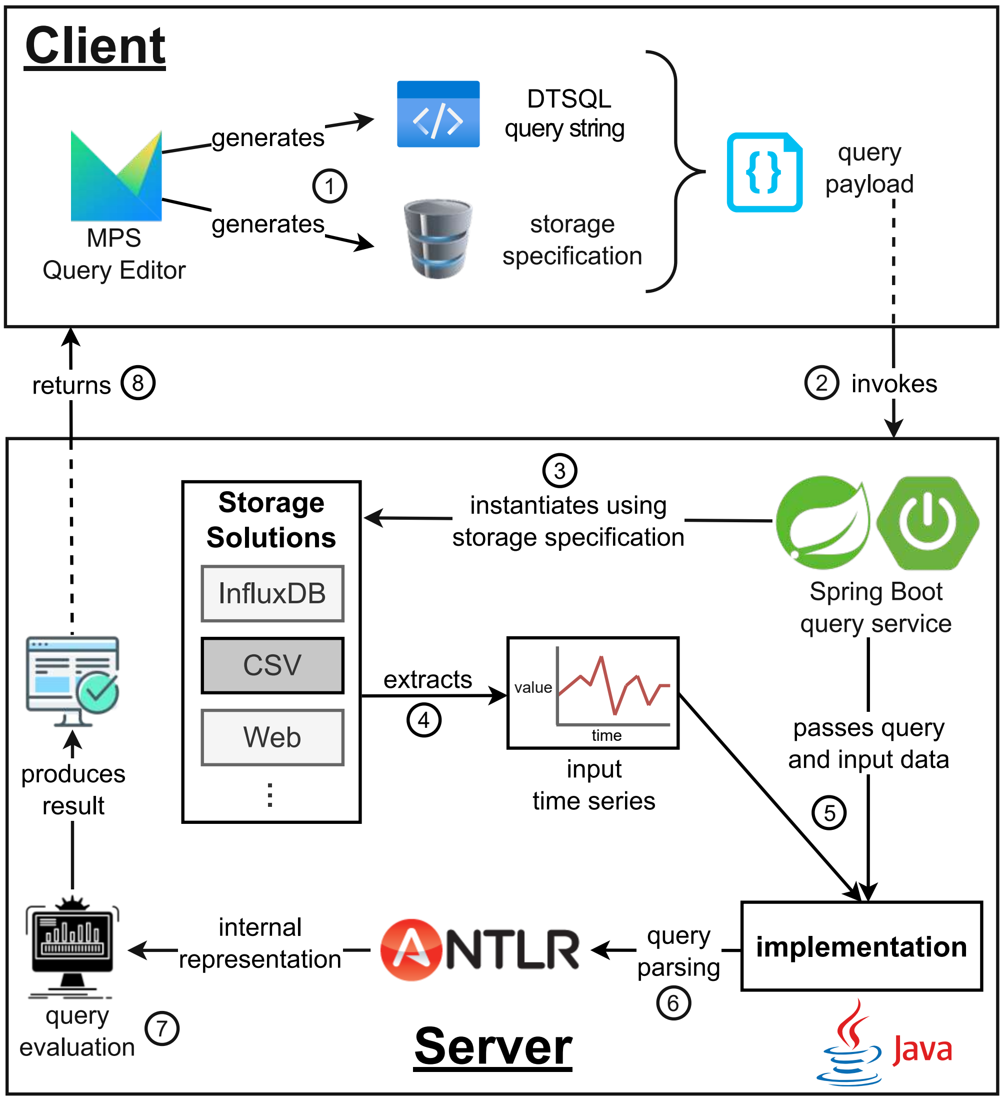


Figure 5.1: Core Components of the DTSQL Reference Implementation

① Users create queries in a semi-visual way by means of a projectional editor (see Chapter 2) provided via JetBrains MPS[1]. Apart from guiding users through the query formulation process, it also allows them to create a storage specification containing information required for reading and transforming time series to be queried. Based on the abstract syntax tree created by the user in the projectional editor, MPS generates the DTSQL query string and storage specification. For further elaborations on and exemplary screenshots of the MPS editor, refer to Section 5.3.

② The query string and storage specification are packed together into a JSON[2] payload and sent to the server application, initiating a query request. This backend service is a REST[3] web service implemented as a Spring Boot[4] application using the Java programming language.

③ By utilizing the storage specification which is part of the request body, the server application instantiates and initializes the specified `StorageService` implementation.

④ Making use of the storage specification again, the initialized storage implementation then reads data in its native format and subsequently transforms it into the canonical representation of a time series to be queried.

⑤ Now, all data required to execute the query contained in the request body has been assembled. The Spring Boot application passes both the time series and the query string to the component which implements the actual query functionality.

⑥ Based on the grammar presented in Section 4.3 and Appendix A.1, a parser generated by ANTLR4[5] is used to convert the query string into an internal representation of a DTSQL query that can be processed effectively.

⑦ As penultimate step, the internal representation of the input query is evaluated to compute the query result. More information on how queries are parsed and how their result objects are computed can be found in Section 5.2.

⑧ Finally, the query result is encoded as web service response, returned to and visualized by the MPS client.

5.2 Query Processing

This section sheds light on the core components of the Java server implementation—i.e., the query evaluation process. Its full source code is available at `https://github.com/dtsql-oss/server`.

[1] `https://www.jetbrains.com/mps/`
[2] JavaScript Object Notation: `https://www.json.org/json-en.html`
[3] Representational State Transfer: a very common architecture for web services exchanging resources
[4] `https://spring.io/projects/spring-boot`
[5] `https://www.antlr.org/index.html`

5.2.1 Parsing and Validation of Query Strings

As already mentioned, ANTLR4 is used to parse DTSQL query strings. The scope of this thesis does not allow to go into detail about ANTLR's architecture and features. The *Definitive ANTLR 4 Reference* [57], written by its creator, provides more exhaustive descriptions than this subsection and additionally presents numerous examples on how to use the parser generator.

In general, ANTLR builds a *parse tree* from the input string, where each node corresponds to a parser rule of the underlying grammar. Afterwards, it offers two interfaces to explore this tree, the *visitor* and the *listener* pattern. With visitors, developers have full control over how the parse tree is traversed. One has to explicitly instruct ANTLR to visit the children of a node, otherwise they are skipped. This provides developers with more flexibility, but also the responsibility to ensure that all relevant nodes are covered during the tree walk. Parse tree listeners, on the other hand, provide an interface reminiscent of SAX[6] parsers. There are *enter* and *exit* callbacks for each node that is encountered by ANTLR's tree walker. Developers may override these callbacks to attach application-specific behavior when processing the various parser-rule-induced nodes.

The DTSQL implementation uses a listener for the majority of language components, due to its simplicity. For recursive structures, however, a custom visitor implementation is employed—e.g., for nested composition operators (see Definition 4.11). By doing so, the nodes processed during the parse tree traversal are read into an internal representation of the DTSQL query string. This allows for an object-oriented query processing and, subsequently, the evaluation (result generation) over given input time series data.

While the grammar ensures basic syntactic validity, the Java listener and visitor implementations also verify some additional constraints regarding semantic integrity:

1. Identifiers of samples and event definitions must be globally unique, i.e., no two identifiable concepts may define the same identifier. More specifically, two events or one event and one sample having the same identifier is invalid.

2. When referencing a concept using an identifier—in a filter, event or composition operator definition—it must have been declared first, using the same identifier.

3. When referencing a concept using an identifier, the target concept must be applicable in the respective context. For instance, a filter argument may be represented by a sample identifer, but not by an event identifier. Analogously, composition operators only accept event identifiers, but no sample identifiers.

4. Query parameters representing points in time must be valid ISO 8601 [58, 59] timestamps. The preferred format is `yyyy-MM-dd'T'HH:mm:ss.zzzXX`. For instance, `2022-09-18T16:42:23.254+01:00` stands for September 18th 2022 at 16:42:23 (and 254 milliseconds) in a timezone which is one hour ahead of UTC (Universal Time Coordinated)—e.g., CET (Central European Time).

[6]Simple API for XML: a standard way of parsing XML documents

If a syntactic or semantic error is encountered during the parsing process, it is halted and a descriptive error message is thrown. Otherwise, if a DTSQL query is successfully parsed into an equivalent internal representation, it as passed onto the evaluation module. The next subsection explains how this module uses that query representation and an input time series to obtain the declaratively specified query result.

5.2.2 Evaluation and Result Generation

The query evaluation process is implemented as outlined in the five-step process from Section 4.2.1. The implementation of steps one (*compute samples*), two (*apply filter*), and five (*assemble result to yield*) are relatively direct Java manifestations of the language semantics presented in Section 4.2.2, Section 4.2.3 and Section 4.2.6, respectively.

While a detailed explanation of the implementation of these steps would not provide more value than the specification of their semantics, it is worth noting that algorithms for high efficiency and numerical accuracy were utilized. For example, all supported summary aggregates (minimum, maximum, sum, standard deviation, count, average) are calculated at once with a single scan of the input time series per unique interval bounds (e.g., all global aggregates have the same interval bounds). The standard deviation is determined using Welford's algorithm [60] which inspects each data point exactly once and significantly reduces imprecision induced by floating point arithmetic. It has been extensively analyzed and compared with other approaches [61, 62]. Similarly, Neumaier's summation algorithm [63] provides a high degree of numerical stability by keeping track of and compensating for numerical errors. It is able to accurately sum up values with considerable differences in magnitude and is therefore an improvement of the previously known Kahan-Babuška method [64, 65].

The methods employed for efficiently capturing intervals specified by event definitions (step three) as well as the selection component (step four) do deserve closer inspection, however. In the following paragraphs, the four most important algorithms linked to these tasks are presented.

Multiple filter event specifications are evaluated at once with only one iteration over the filtered input time series, as shown in Algorithm 5.1. The idea of this method is similar to the *single scan coalescing* algorithm proposed in [66]. In the context of temporal databases, the *coalesce* operation is comparable to deleting tuples with identical values from a conventional database (duplicate elimination). It merges value-equivalent tuples that have adjacent or overlapping time periods [67]. The process displayed in Algorithm 5.1 differs from the single-scan coalesce algorithm in that it coalesces the data points into intervals based on whether they satisfy the respective event definition, rather than them having equivalent values. It keeps track of the point in time starting from which an event is continuously satisfied using the map m (lines 3, 12–13). If an event is not satisfied anymore, then the interval described by the current point in time and the memorized one is added to the set of detected intervals (lines 2, 16–18). There is also a special case that ends a period—even though the event is still satisfied—if the end of the input data is reached (lines 8–10).

Algorithm 5.1: Interval Detection of Filter Events

```
   Input:  filtered time series $\overline{\Psi} = \langle p_1, \dots, p_n \rangle$,
           filter event functions $\mathcal{E}_f = \{e_1, \dots, e_m\}$
   Output: set of intervals $\Pi$ over $\overline{\Psi}$ such that eval_e($e$, ${}_i\pi_j$) $=$ true for at least
           one $e \in \mathcal{E}_f$ for all ${}_i\pi_j \in \Pi$

 1 function detectFilterEventIntervals($\overline{\Psi}$, $\mathcal{E}_f$):
 2     $\Pi \leftarrow \emptyset$ // set of detected intervals
 3     $m \leftarrow \{(\mathcal{E}_f \rightarrow \mathbb{Z})\}$ // maps events to starts of open periods
 4     foreach $p_i \in \overline{\Psi}$ do
 5         foreach $e \in \mathcal{E}_f$ do  // event $e$ is equivalent to filter $f$
 6             if eval_f($f, p_i$) $=$ true then
 7                 if $m[e] \neq$ null then
 8                     if $i = n$ then
 9                         $\Pi = \Pi \cup \left\{{}_{m[e]}\pi_{\texttt{dpt}(p_i)}\right\}$
10                         $m[e] \leftarrow$ null
11                     end
12                 else
13                     $m[e] \leftarrow$ dpt($p_i$)
14                 end
15             else
16                 if $m[e] \neq$ null then
17                     $\Pi = \Pi \cup \left\{{}_{m[e]}\pi_{\texttt{dpt}(p_i)}\right\}$
18                     $m[e] \leftarrow$ null
19                 end
20             end
21         end
22     end
23     return $\Pi$
24 end
```

In the category of complex events, the procedure for capturing periods of constant events is outlined in Algorithm 5.2. At first, an over-approximation of interval candidates is determined by capturing periods within which the instantaneous rate of change is in the range $\pm t$, where t is an internal sensitivity threshold which depends on the input data (line 2). Afterwards, the slope of the regression line in the interval candidates (lines 5–6) as well as the deviation from the average value (lines 12–13) are verified. If both checks succeed, the respective period is added to the result set (line 14). It should be mentioned that, strictly speaking, this procedure does not guarantee maximality. Furthermore, line

2 only yields a “true” over-approximation if the data-dependent sensitivity threshold t is large enough (but not so large that no interval candidate satisfies `regc` and `devc`). In order to ensure that the maximality criterion holds, an efficient neighborhood search—in the sense of enlarging and/or shrinking candidate intervals to the left and/or right until its size is maximal—would have to be devised.

Algorithm 5.2: Interval Detection of Constant Events

```
Input:  filtered time series $\overline{\Psi} = \langle p_1, \ldots, p_n \rangle$,
        constant event $e = \texttt{const}(s, d)$
Output: interval set $\Pi$ over $\overline{\Psi}$ such that $\texttt{eval\_e}(e,\ {}_i\pi_j) = \texttt{true}$ for all ${}_i\pi_j \in \Pi$

1  function detectConstantEventIntervals($\overline{\Psi}$, $s$, $d$):
       // interval candidates, ($\overline{\Psi}'$: derivative)
2      $c \leftarrow \texttt{detectFilterEventIntervals}(\overline{\Psi}', \{\texttt{around\_abs}_p(0.0,\ t)\})$
3      $r \leftarrow \emptyset$ // intervals satisfying regc constraint
4      foreach ${}_i\pi_j \in c$ do
5          $\beta \leftarrow \texttt{linearRegressionSlope}(\texttt{ps}_{\overline{\Psi}}({}_i\pi_j))$
6          if $|\beta| \cdot 100 \leq s$ then
7              $r \leftarrow r \cup \{{}_i\pi_j\}$
8          end
9      end
10     $rd \leftarrow \emptyset$ // intervals satisfying regc and devc constraints
11     foreach ${}_i\pi_j \in r$ do
12         $a \leftarrow \texttt{eval\_s}(\texttt{avg}_{\texttt{ps}_{\overline{\Psi}}({}_i\pi_j)})$
13         if $\forall p \in \texttt{ps}_{\overline{\Psi}}({}_i\pi_j){:}\ \left(\frac{|\texttt{dpv}(p)-a|}{|a|} \cdot 100 \leq d\right)$ then
14             $rd \leftarrow rd \cup \{{}_i\pi_j\}$
15         end
16     end
17     return $rd$
18 end
```

The intervals specified by monotonic events are determined similarly. In Algorithm 5.3, the process of detecting monotonically increasing intervals is depicted. First, all intervals satisfying the constraint regarding the instantaneous rate of change are determined (line 2). This ensures maximality because, due to the maximality of `detectFilterEventIntervals`, there can be no longer intervals where `ratc` continuously holds. Afterwards, the relative change in value between the starting and ending points of the just determined intervals is verified (lines 5–6). If this check succeeds as well, the respective period is added to the result set. This algorithm works analogously for decreasing intervals, with slight modifications: The filter argument in line 2 would need to be changed to “$\{\texttt{lt}_p(\frac{t}{100})\}$” and the condition in line 6 to “$c \leq 0$ *and* $|c| \in [l, u]$”.

Algorithm 5.3: Interval Detection of Monotonic Increase Events

```
Input:  filtered time series $\overline{\Psi} = \langle p_1, \ldots, p_n \rangle$,
        increase event $e = \texttt{increase}(l, u, t)$
Output: interval set $\Pi$ over $\overline{\Psi}$ such that $\texttt{eval\_e}(e, {}_i\pi_j) = \texttt{true}$ for all ${}_i\pi_j \in \Pi$

1  function detectIncreaseEventIntervals($\overline{\Psi}$, $l$, $u$, $t$):
       // intervals satisfying ratc constraint ($\overline{\Psi}'$: derivative)
2      $r \leftarrow \texttt{detectFilterEventIntervals}(\overline{\Psi}', \{\texttt{gt}_p(-\frac{t}{100})\})$
3      $rd \leftarrow \emptyset$ // intervals satisfying ratc and difc constraints
4      foreach ${}_i\pi_j \in r$ do
5          $c \leftarrow \frac{\texttt{dpv}(p_j) - \texttt{dpv}(p_i)}{|\texttt{dpv}(p_i)|} \cdot 100$
6          if $c \geq 0$ and $c \in [l, u]$ then
7              $rd \leftarrow rd \cup \{{}_i\pi_j\}$
8          end
9      end
10     return $rd$
11 end
```

Finally, Algorithm 5.4 demonstrates how the intervals resulting from event specifications—detected using the preceding three algorithms—are used to create composite events, i.e., event sequences. A selection component consists of either a `precedes` or a `follows` root composition operator. Since the `follows` relation is the inverse of the `precedes` relation, the evaluation of `follows` is reduced to evaluating `precedes` operators with swapped arguments (lines 2–5). In lines 9–14, the set of intervals to consider when looking for event sequences corresponding to the selection component is assembled. Should an operator not represent an event, but be a nested (recursive) composition operator, then all intervals represented by it are determined by invoking the algorithm recursively (lines 11 and 14). Afterwards, pairs of periods p_i and p_j—where p_j starts before p_i—are examined with respect to whether they satisfy the conditions of the `precedes` composition operator (line 19). If they do, then the corresponding composite (merged) interval—made up of the start of p_j and the end of p_i—is added to the result set (line 20).

After evaluating the selection component, the overall query result is generated and returned to the user as specified in Definition 4.28 (Semantics of DTSQL Yield Component). As mentioned before, there is no additional value in providing a concrete pseudocode implementing the yield semantics.

Algorithm 5.4: Evaluation of Selection Component

```
Input:  composition operator $\omega = opt(\boxplus_1, \boxplus_2)$ with
        $opt \in \{\texttt{precedes}, \texttt{follows}\}$,
        set of intervals $\Pi$ detected from event definitions
Output: set of intervals specified by input, i.e., value of $\texttt{eval\_o}(opt(\boxplus_1, \boxplus_2))$

function evaluateSelection ($opt$, $\boxplus_1$, $\boxplus_2$, $\Pi$):
    if $opt = \texttt{precedes}$ then
        return evaluatePrecedes($\boxplus_1, \boxplus_2$)
    else
        return evaluatePrecedes($\boxplus_2, \boxplus_1$)
    end
end
function evaluatePrecedes ($\boxplus_1$, $\boxplus_2$, $\Pi$):
    $\widehat{\Pi} \leftarrow \Pi$ // set of intervals to examine
    if $\boxplus_1 = \widetilde{opt}_1(\widetilde{\boxplus}_1, \widetilde{\boxplus}_2)$ then  // recursive first argument
        $\widehat{\Pi} \leftarrow \widehat{\Pi} \cup$ evaluateSelection($\widetilde{opt}_1$, $\widetilde{\boxplus}_1$, $\widetilde{\boxplus}_2$, $\Pi$)  // adds eval_a($\boxplus_1$)
    end
    if $\boxplus_2 = \widetilde{opt}_2(\widetilde{\boxplus}_3, \widetilde{\boxplus}_4)$ then  // recursive second argument
        $\widehat{\Pi} \leftarrow \widehat{\Pi} \cup$ evaluateSelection($\widetilde{opt}_2$, $\widetilde{\boxplus}_3$, $\widetilde{\boxplus}_4$, $\Pi$)  // adds eval_a($\boxplus_2$)
    end
    $\Pi' \leftarrow \emptyset$ // set of chosen intervals
    foreach $p_i \in \widehat{\Pi}$, $i \in \{1, \dots, |\widehat{\Pi}|\}$ do
        foreach $p_j \in \widehat{\Pi}$, $j \in \{1, \dots, i-1\}$ do
            if memc($p_j$, $p_i$) = true and relc($p_j$, $p_i$, precedes) = true and
             seqc($p_j$, $p_i$, precedes) = true then
                $\Pi' \leftarrow \Pi' \cup \left\{ {}_{\texttt{intstart}(p_j)}\pi_{\texttt{intend}(p_i)} \right\}$
            end
        end
    end
    return $\Pi'$
end
```

5.3 Client Environment

This section presents how JetBrains MPS was used to create a client environment that assists users in formulating and executing DTSQL queries. As already noted in Section 2.3.2, due to space restrictions, it is not possible to give in-depth explanations of how to use MPS. Therefore, this section only showcases the resulting client environment. Its implementation is available at `https://github.com/dtsql-oss/query-client`. The interested reader may also find MPS-related resources in the official documentation[7] or relevant textbooks, e.g., [68, 69, 52].

5.3.1 Query Formulation

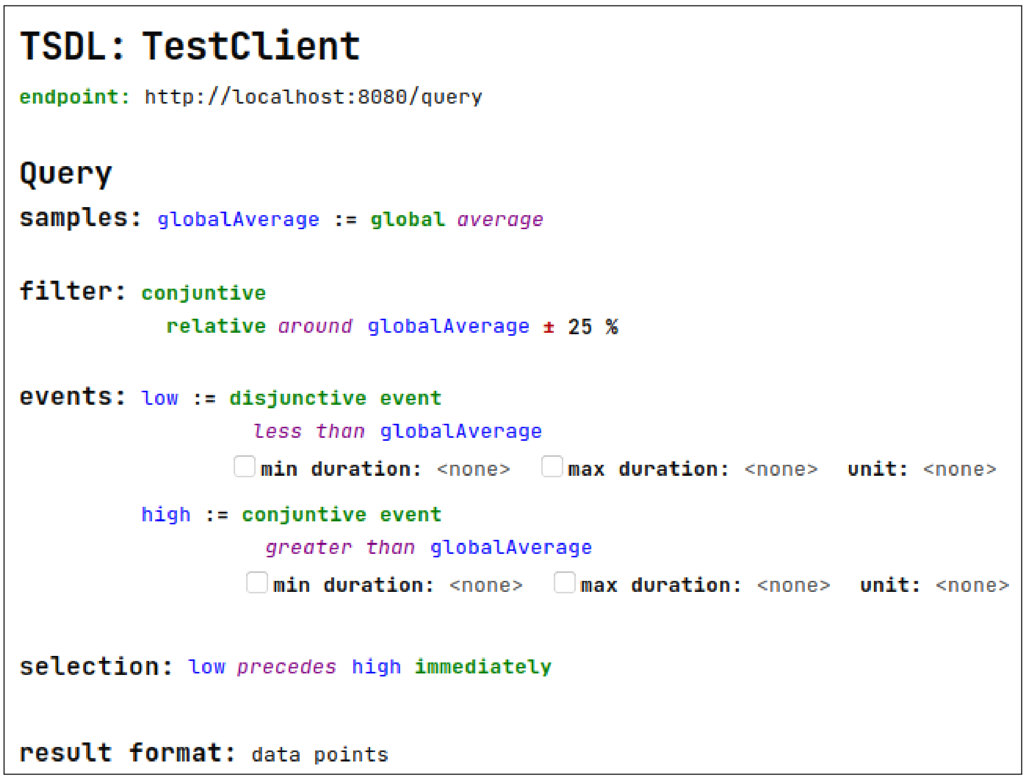
MPS offers purpose-tailored ways to model the structure (syntax) of a DSL, define a projectional editor for writing programs in it, express (syntactic and semantic) validity constraints, implement context-actions as well as add code generation from a valid DSL program. While MPS provides much more than listed in this enumeration, these are the main features used by the DTSQL client environment.

The especially designed projectional editor guides users through the DTSQL query creation process. It provides context-sensitive auto-complete functionality as well as interactive elements such as checkboxes, relieving users of memorizing the exact syntax of every language feature by heart. Furthermore, the use of colors for different query elements makes it easier to grasp the intention of a query. As an example, the result of modelling the query from Listing 4.2 in MPS is depicted in Figure 5.2.

Figure 5.3 demonstrates in more detail how the editor supports users in formulating DTSQL queries. For instance, it offers available types of samples (a), event functions (c) and result (d) types. Furthermore, it suggests negating event or filter functions (b) using *intentions*. Similarly, many parameters in a DTSQL query may either be literal values or reference samples (e). In the latter case, the auto-complete menu only offers valid, previously defined sample identifiers (f). Finally, event sequences might be equipped with time-gap constraints (g). If such a constraint is present, the editor exclusively offers time units that are supported by DTSQL (h).

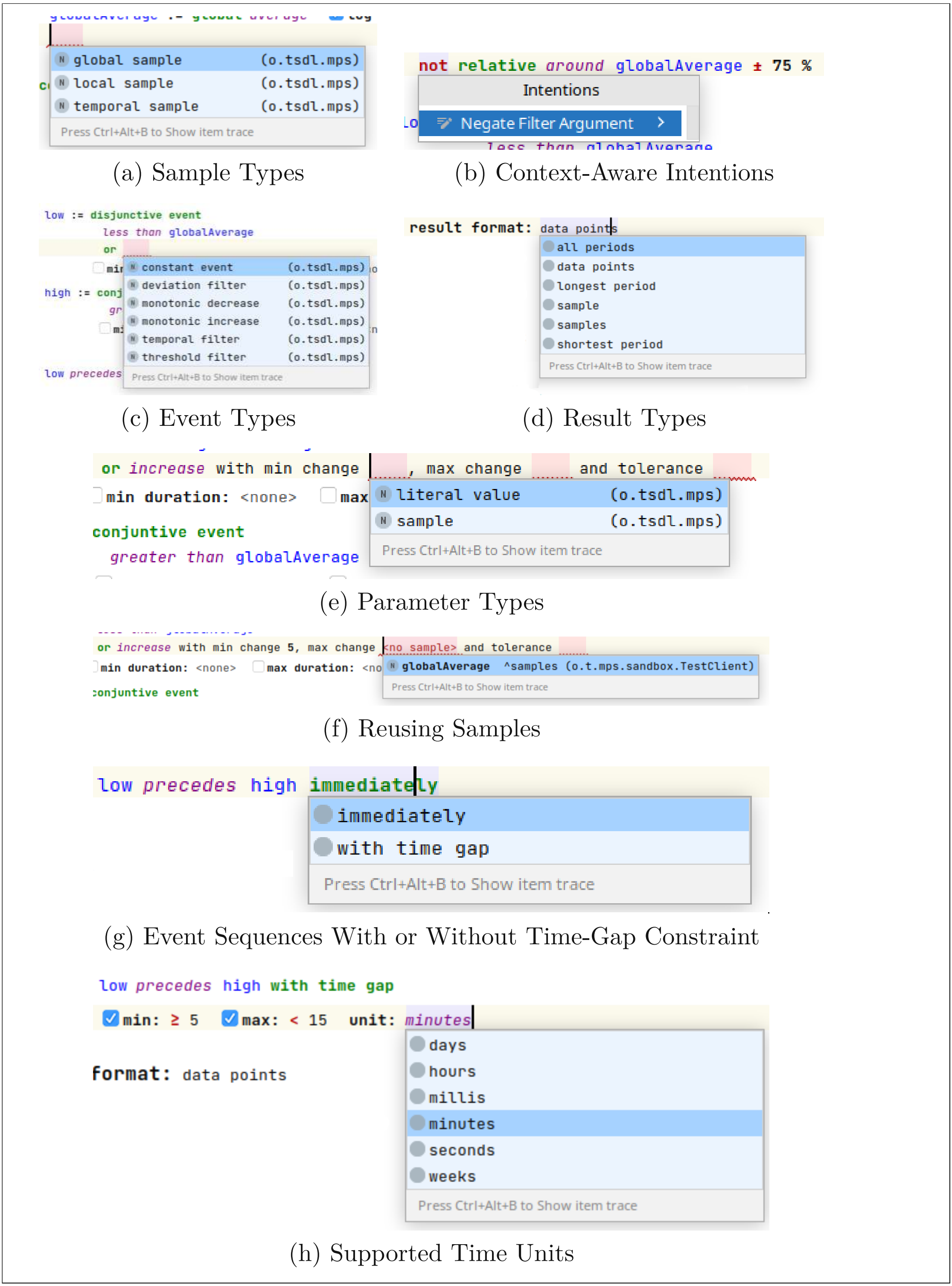
In addition to the above ways the projectional editor helps users create DTSQL queries, it also makes use of MPS's capabilities in formulating constraints for DSLs. This has the positive effect of issuing domain-specific and clear error messages during the query design process, rather than receiving abstract error messages generated by the ANTLR parser during the query evaluation process. Five instances of such domain-specific errors are depicted in Figure 5.4. It shows that sample identifiers must not start with letters and that timestamps must be valid with respect to the ISO 8601 standard (a). Furthermore, event durations must be expressed in a concrete unit and must not be negative (b). Ultimately, a yield component returning the computed values of samples must be accompanied by at least one sample identifier (c).

[7] `https://www.jetbrains.com/help/mps/mps-user-s-guide.html`



```
TSDL: TestClient
endpoint: http://localhost:8080/query

Query
samples: globalAverage := global average

filter: conjuntive
          relative around globalAverage ± 25 %

events: low := disjunctive event
                 less than globalAverage
               ☐min duration: <none>   ☐max duration: <none>   unit: <none>
        high := conjuntive event
                  greater than globalAverage
                ☐min duration: <none>   ☐max duration: <none>   unit: <none>

selection: low precedes high immediately

result format: data points
```

Figure 5.2: MPS Representation of DTSQL Query From Listing 4.2



```
global sample      (o.tsdl.mps)
local sample       (o.tsdl.mps)
temporal sample    (o.tsdl.mps)
Press Ctrl+Alt+B to Show item trace
```

(a) Sample Types

```
not relative around globalAverage ± 75 %
Intentions
Negate Filter Argument
```

(b) Context-Aware Intentions

```
low := disjunctive event
    less than globalAverage
    or
high := conj
low precedes
constant event      (o.tsdl.mps)
deviation filter    (o.tsdl.mps)
monotonic decrease  (o.tsdl.mps)
monotonic increase  (o.tsdl.mps)
temporal filter     (o.tsdl.mps)
threshold filter    (o.tsdl.mps)
Press Ctrl+Alt+B to Show item trace
```

(c) Event Types

```
result format: data points
all periods
data points
longest period
sample
samples
shortest period
Press Ctrl+Alt+B to Show item trace
```

(d) Result Types

```
or increase with min change , max change  and tolerance
min duration: <none>   max
conjuntive event
  greater than globalAverage
literal value   (o.tsdl.mps)
sample          (o.tsdl.mps)
Press Ctrl+Alt+B to Show item trace
```

(e) Parameter Types

```
or increase with min change 5, max change <no sample> and tolerance
min duration: <none>   max duration: <no
conjuntive event
globalAverage  ^samples (o.t.mps.sandbox.TestClient)
Press Ctrl+Alt+B to Show item trace
```

(f) Reusing Samples

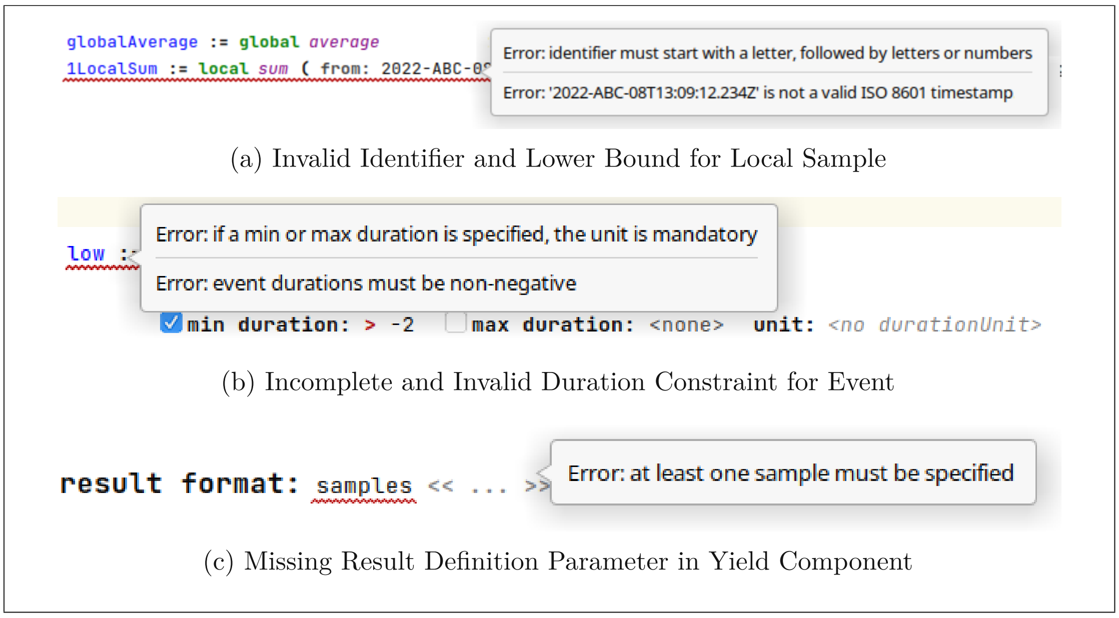
```
low precedes high immediately
immediately
with time gap
Press Ctrl+Alt+B to Show item trace
```

(g) Event Sequences With or Without Time-Gap Constraint

```
low precedes high with time gap
☑min: ≥ 5  ☑max: < 15  unit: minutes
format: data points
days
hours
millis
minutes
seconds
weeks
Press Ctrl+Alt+B to Show item trace
```

(h) Supported Time Units

Figure 5.3: Selected Aspects of Interactive MPS Editor for DTSQL Queries



```
globalAverage := global average
1LocalSum := local sum ( from: 2022-ABC-0
```

Error: identifier must start with a letter, followed by letters or numbers

Error: '2022-ABC-08T13:09:12.234Z' is not a valid ISO 8601 timestamp

(a) Invalid Identifier and Lower Bound for Local Sample

```
low :
```

Error: if a min or max duration is specified, the unit is mandatory

Error: event durations must be non-negative

```
☑min duration: > -2   ☐max duration: <none>   unit: <no durationUnit>
```

(b) Incomplete and Invalid Duration Constraint for Event

```
result format: samples << ... >>
```

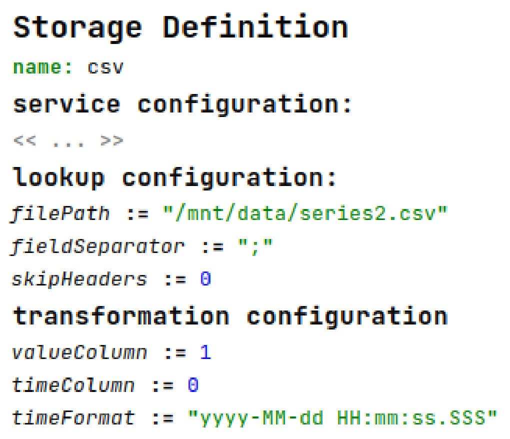
Error: at least one sample must be specified

(c) Missing Result Definition Parameter in Yield Component

Figure 5.4: DTSQL Query Validation Provided by the MPS Editor

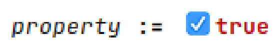
5.3.2 Query Execution

Using code generation, MPS makes it possible to build and execute a Java application from a given DSL program. In order to create such a program that executes a modelled DTSQL query, a storage configuration that defines how to obtain an input time series is also required. Figure 5.5 shows how the client environment enables users to define a storage specification. In (a), the storage definition for a time series deposited in a CSV file `series2.csv` is depicted—its data may also be found in Appendix A.2. This storage configuration features various configuration properties necessary to parse the CSV time series into the canonical representation—e.g., field separator, number of lines to skip and timestamp format. Generally, storage configurations may contain boolean (b), number (integer or float) and string properties (c).



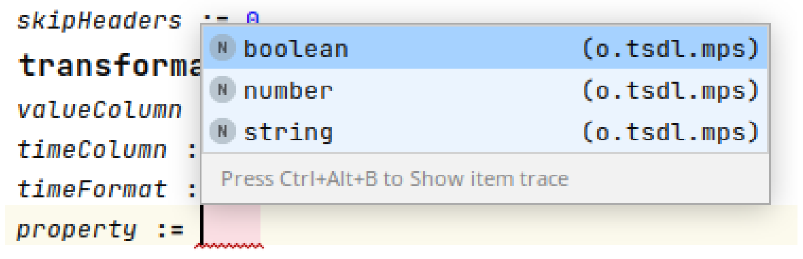
```
Storage Definition
name: csv
service configuration:
<< ... >>
lookup configuration:
filePath := "/mnt/data/series2.csv"
fieldSeparator := ";"
skipHeaders := 0
transformation configuration
valueColumn := 1
timeColumn := 0
timeFormat := "yyyy-MM-dd HH:mm:ss.SSS"
```

(a) CSV Storage Specification



```
property := ✓ true
```

(b) Projection of Boolean Properties



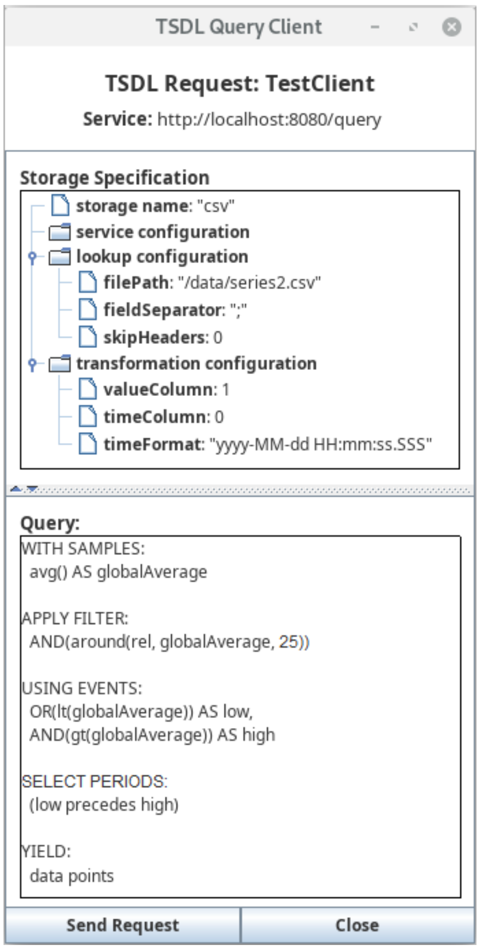
```
skipHeaders := 0
transforma
valueColumn
timeColumn :
timeFormat :
property :=
```

```
N boolean          (o.tsdl.mps)
N number           (o.tsdl.mps)
N string           (o.tsdl.mps)
Press Ctrl+Alt+B to Show item trace
```

(c) Supported Storage Property Types

Figure 5.5: MPS Representation of Storage Specifications

Combining this storage specification with the previously explained query model, the client environment is able to generate the Java application shown in Figure 5.6. It gives a hierarchical overview of the storage specification as well as the generated DTSQL query string (a). In addition to that, it visualizes the input time series that was obtained using the storage specification (b)—again, see Appendix A.2 for the concrete data.



TSDL Request: TestClient

Service: http://localhost:8080/query

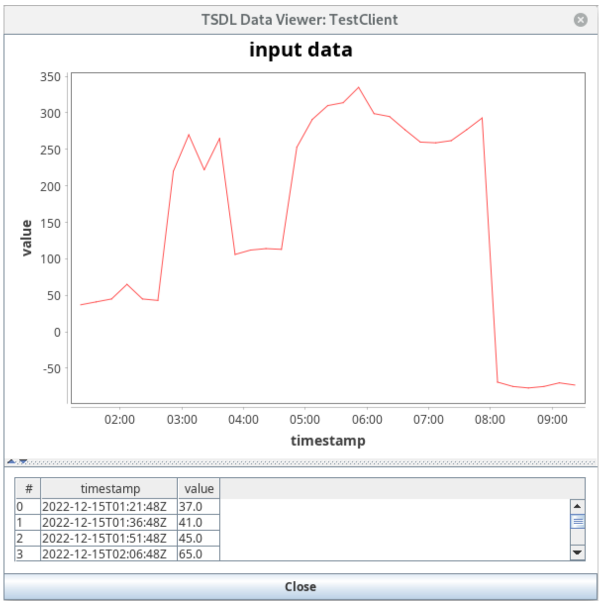
Storage Specification

- **storage name**: "csv"
- **service configuration**
- **lookup configuration**
  - **filePath**: "/data/series2.csv"
  - **fieldSeparator**: ";"
  - **skipHeaders**: 0
- **transformation configuration**
  - **valueColumn**: 1
  - **timeColumn**: 0
  - **timeFormat**: "yyyy-MM-dd HH:mm:ss.SSS"

Query:

```
WITH SAMPLES:
  avg() AS globalAverage

APPLY FILTER:
  AND(around(rel, globalAverage, 25))

USING EVENTS:
  OR(lt(globalAverage)) AS low,
  AND(gt(globalAverage)) AS high

SELECT PERIODS:
  (low precedes high)

YIELD:
  data points
```

(a) Storage Specification and Generated DTSQL Query

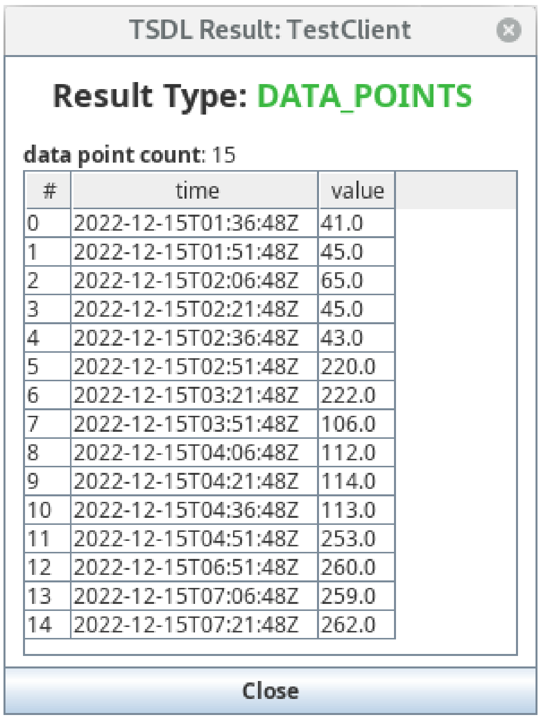


#	timestamp	value
0	2022-12-15T01:21:48Z	37.0
1	2022-12-15T01:36:48Z	41.0
2	2022-12-15T01:51:48Z	45.0
3	2022-12-15T02:06:48Z	65.0

(b) Visualization of Input Time Series

Figure 5.6: Java Application Generated by MPS

Finally, after executing the generated DTSQL query shown in Figure 5.6a, its result is visualized in a separate window. There are specific views for each supported result type (yield format). Two examples are given in Figure 5.7. While the result depicted in Figure 5.7a corresponds to the query from Figure 5.6a, Figure 5.7b shows a version of the same query which has been adapted to return all detected periods. As the figures illustrate, the data point view shows the timestamps and values of the returned data points. On the other hand, the period set view displays the number of detected periods together with their respective start and end timestamps.

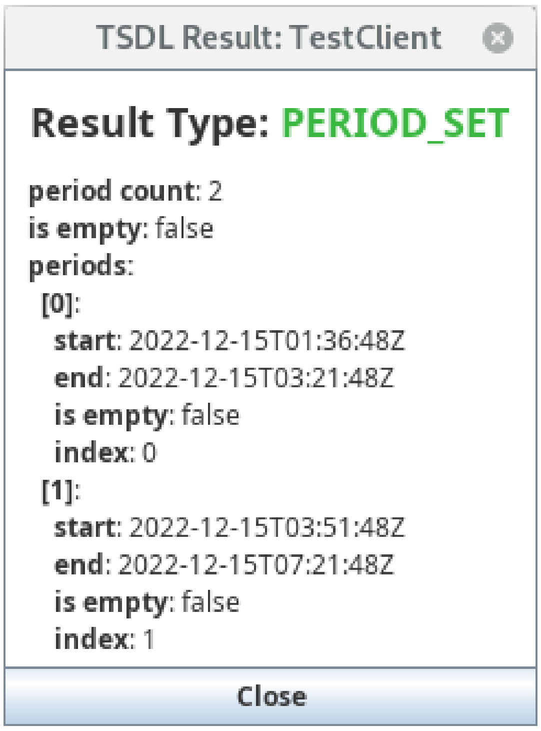


Result Type: DATA_POINTS

data point count: 15

#	time	value
0	2022-12-15T01:36:48Z	41.0
1	2022-12-15T01:51:48Z	45.0
2	2022-12-15T02:06:48Z	65.0
3	2022-12-15T02:21:48Z	45.0
4	2022-12-15T02:36:48Z	43.0
5	2022-12-15T02:51:48Z	220.0
6	2022-12-15T03:21:48Z	222.0
7	2022-12-15T03:51:48Z	106.0
8	2022-12-15T04:06:48Z	112.0
9	2022-12-15T04:21:48Z	114.0
10	2022-12-15T04:36:48Z	113.0
11	2022-12-15T04:51:48Z	253.0
12	2022-12-15T06:51:48Z	260.0
13	2022-12-15T07:06:48Z	259.0
14	2022-12-15T07:21:48Z	262.0

(a) Result View for `YIELD: data points`



Result Type: PERIOD_SET

```
period count: 2
is empty: false
periods:
 [0]:
   start: 2022-12-15T01:36:48Z
   end: 2022-12-15T03:21:48Z
   is empty: false
   index: 0
 [1]:
   start: 2022-12-15T03:51:48Z
   end: 2022-12-15T07:21:48Z
   is empty: false
   index: 1
```

(b) Result View for `YIELD: all periods`

Figure 5.7: Query Result Views in Java Application for Two Yield Components

CHAPTER 6

Evaluation

This chapter evaluates the core contributions of this thesis. First, it gauges whether the use cases described in Chapter 3 can be expressed using DTSQL, as it has been specified in Chapter 4. Afterwards, a quantitative as well as qualitative assessment examines the performance of the reference implementation, as presented in Chapter 5. Lastly, the overall system's usefulness is discussed from the perspective of the domain experts accompanying this work.

6.1 Coverage of Practical Use Cases

The language specification from Chapter 4 has turned out to be expressive enough to cover all the use cases identified in Chapter 3. In order to demonstrate this, the following subsections will provide a query for each use case.

6.1.1 UC1: Global Aggregates

This use case is concerned with calculating aggregate functions that represent the average, count, maximum, minimum, standard deviation and sum of all values in a time series. The query depicted by Listing 6.1 shows how to capture these values.

```
WITH SAMPLES:
  avg() AS myAverage,
  count() AS myCount,
  max() AS myMax,
  min() AS myMin,
  stddev() AS myStdDev,
  sum() AS mySum
YIELD:
  samples myAvg, myStdDev, myMin, myMax, mySum, myCount
```

Listing 6.1: DTSQL Query Expressing "UC1: Global Aggregates"

6.1.2 UC2: Local Aggregates

Very similar to the first use case, this one necessitates the very same aggregate functions, but only in sub-intervals of a time series. A query showcasing this is illustrated in Listing 6.2. Notice that DTSQL allows the lower and/or upper bound of a local aggregate to be left unspecified, denoted by empty string literal (`""`). In that case, the start/end of the overall time series is used as interval bound(s).

```
WITH SAMPLES:
  avg("2017-09-09T10:00:00Z", "") AS localAverage,
  count("", "2019-09-04T10:00:00Z") AS localCount,
  max("2017-09-09T10:00:00Z", "2019-09-04T10:00:00Z") AS localMax,
  min("2017-09-09T10:00:00Z", "2019-09-04T10:00:00Z") AS localMin,
  stddev("2017-09-09T10:00:00Z", "2019-09-04T10:00:00Z") AS localStdDev,
  sum("2017-09-09T10:00:00Z", "2019-09-04T10:00:00Z") AS localSum
YIELD:
  samples localAverage, localStdDev, localMin, localMax, localSum, localCount
```

Listing 6.2: DTSQL Query Expressing “UC2: Local Aggregates”

6.1.3 UC3: Temporal Aggregates

In this use case, the same aggregate functions as before are calculated, but over the temporal instead of the value dimension. In other words, the duration of a number of intervals—each specified in the format `"start/end"`—is aggregated and returned in a time unit supported by DTSQL. This is exemplified by the query in Listing 6.3.

```
WITH SAMPLES:
  avg_t(minutes, "2017-09-09T10:00:00Z/2019-09-04T10:00:00Z",
                 "2018-06-11T10:00:00Z/2019-01-08T10:00:00Z",
                 "2019-06-27T10:00:00Z/2019-09-12T10:00:00Z") AS tempAverage,
  count_t("2017-09-09T10:00:00Z/2019-09-04T10:00:00Z",
          "2018-06-11T10:00:00Z/2019-01-08T10:00:00Z",
          "2019-06-27T10:00:00Z/2019-09-12T10:00:00Z") AS tempCount,
  max_t(hours, "2017-09-09T10:00:00Z/2019-09-04T10:00:00Z",
               "2018-06-11T10:00:00Z/2019-01-08T10:00:00Z",
               "2019-06-27T10:00:00Z/2019-09-12T10:00:00Z") AS tempMax,
  min_t(seconds, "2017-09-09T10:00:00Z/2019-09-04T10:00:00Z",
                 "2018-06-11T10:00:00Z/2019-01-08T10:00:00Z",
                 "2019-06-27T10:00:00Z/2019-09-12T10:00:00Z") AS tempMin,
  stddev_t(days, "2017-09-09T10:00:00Z/2019-09-04T10:00:00Z",
                 "2018-06-11T10:00:00Z/2019-01-08T10:00:00Z",
                 "2019-06-27T10:00:00Z/2019-09-12T10:00:00Z") AS tempStdDev,
  sum_t(weeks, "2017-09-09T10:00:00Z/2019-09-04T10:00:00Z",
               "2018-06-11T10:00:00Z/2019-01-08T10:00:00Z",
               "2019-06-27T10:00:00Z/2019-09-12T10:00:00Z") AS tempSum
YIELD:
  samples tempAverage, tempStdDev, tempMin, tempMax, tempSum, tempCount
```

Listing 6.3: DTSQL Query Expressing “UC3: Temporal Aggregates”

6.1.4 UC4: Numerical Integral

This use case represents the calculation of a numerical integral by computing the area under the curve described by a time series with linear interpolation between pairwise adjacent data points. In DTSQL, this is expressed the same way as global and local aggregates—as demonstrated in Listing 6.4.

```
WITH SAMPLES:
  integral() AS globalIntegral,
  integral("2017-09-08T10:00:00Z", "2017-10-12T10:00:00Z") AS localIntegral,
YIELD:
  samples globalIntegral, localIntegral
```

Listing 6.4: DTSQL Query Expressing “UC4: Numerical Integral”

6.1.5 UC5: Threshold Filters

Threshold filters are used to filter out data points with value components that are higher or lower than a certain threshold. Listing 6.5 displays a query that only keeps data points whose values are greater than the global average, but not greater than 875.75.

```
WITH SAMPLES:
  avg() AS globalAvg
APPLY FILTER:
  AND(gt(globalAvg), NOT(gt(875.75)))
YIELD:
  data points
```

Listing 6.5: DTSQL Query Expressing “UC5: Threshold Filters”

6.1.6 UC6: Temporal Filters

Temporal filters are conceptually very similar to threshold filters. However, they filter out data points based on whether their time component is before or after a certain point in time. The query depicted in Listing 6.6 keeps only data points which were recorded either before `2018-01-13T10:00:00Z` or not before `2019-08-07T10:00:00Z`.

```
APPLY FILTER:
  OR(before("2018-01-13T10:00:00Z"), NOT(before("2019-08-07T10:00:00Z")))
YIELD:
  data points
```

Listing 6.6: DTSQL Query Expressing “UC6: Temporal Filters”

6.1.7 UC7: Threshold Events

This use case aims to detect intervals during which the data point values are consistently above or below a certain threshold. A corresponding query that captures all periods with values above the global average is given in Listing 6.7.

```
WITH SAMPLES:
  avg() AS globalAvg
USING EVENTS:
  AND(gt(globalAvg)) AS high
YIELD:
  all periods
```

Listing 6.7: DTSQL Query Expressing “UC7: Threshold Events”

6.1.8 UC8: Deviation Events

Using deviation events, it is possible to detect periods where the values of recorded data points are within a specifiable (relative or absolute) range of an arbitrary reference value. The query in Listing 6.8 first applies a temporal filter and then detects the longest period with values that are in a ±2.5 % range of a local average.

```
WITH SAMPLES:
  avg("2018-06-04T10:00:00Z", "2019-01-22T10:00:00Z") AS localAvg
APPLY FILTER:
  AND(NOT(before("2018-06-04T10:00:00Z")),NOT(after("2019-01-22T10:00:00Z")))
USING EVENTS:
  AND(around(rel, localAvg, 2.5)) AS deviationEvent
YIELD:
  longest period
```

Listing 6.8: DTSQL Query Expressing “UC8: Deviation Events”

6.1.9 UC9: Constant Events

This use case is concerned with detecting periods that represent (approximately) constant signals, according to two tolerance parameters. In Listing 6.9, constant periods are detected which define a regression line whose slope is not greater than 10 % and whose values reside in a ±5 % range of its average.

```
USING EVENTS:
  AND(const(10, 5)) AS constantEvent
YIELD:
  all periods
```

Listing 6.9: DTSQL Query Expressing “UC9: Constant Events”

6.1.10 UC10: Monotonic Events

Monotonic events describe periods of increase or decrease. In DTSQL, they are defined by a minimum and maximum relative change occurring during a period as well as a value that specifies a tolerance against temporary decreases/increases. For instance, the query in Listing 6.10 demonstrates how to detect periods of moderate increase (5.25–25.5 %) which at no point have an instantaneous rate of change lower than -600 %.

```
USING EVENTS:
  AND(increase(5.25, 25.5, 600.0)) AS increaseEvent
YIELD:
  all periods
```

Listing 6.10: DTSQL Query Expressing “UC10: Monotonic Events”

6.1.11 UC11: Duration Constraints for Events

There are cases where events are only relevant if their duration is within a specific range. Listing 6.11 modifies the query from Listing 6.10 to include only those periods which are more than two, but not more than five weeks long.

```
USING EVENTS:
  AND(increase(5.25, 25.5, 600.0)) FOR (2,5] weeks AS increaseEvent
YIELD:
  all periods
```

Listing 6.11: DTSQL Query Expressing “UC11: Duration Constraints for Events”

6.1.12 UC12: Binary Event Sequences

In this use case, it is required to express temporal relations between periods detected as a result of event definitions. For example, a query capturing periods where a period with values higher than the global average appears before a period with values lower than the global average is presented in Listing 6.12. Both kinds of periods should exhibit a length of at least 20 days.

```
WITH SAMPLES:
  avg() AS globalAvg
USING EVENTS:
  AND(lt(globalAvg)) FOR [20,] days AS low,
  AND(gt(globalAvg)) FOR [20,] days AS high
SELECT PERIODS:
  (high precedes low)
YIELD:
  all periods
```

Listing 6.12: DTSQL Query Expressing “UC12: Binary Event Sequences”

6.1.13 UC13: Time Tolerance for Event Sequences

This use case has the purpose of formulating time-gap constraints for temporal relations, i.e., a minimum and/or maximum amount of time that may pass between periods in an event sequence. A query demonstrating a time gap of 30 to 100 days between a constant and an increasing period is shown in Listing 6.13.

```
USING EVENTS:
  AND(const(10.0, 5.0)) AS constantEvent,
  AND(increase(5.25, 25.5, 600.0)) FOR (2,5] weeks AS increaseEvent
SELECT PERIODS:
  (constantEvent precedes increaseEvent WITHIN [30,100] days)
YIELD:
  all periods
```

Listing 6.13: DTSQL Query Expressing "UC13: Time Tolerance for Event Sequences"

6.1.14 UC14: N-Ary Event Sequences

Temporal relations can also be formulated recursively to express n-ary event sequences. The query depicted in Listing 6.14 first applies a temporal filter, and then detects periods where a constant period appears before a switch from values above to below the global average, with not more than two days in between.

```
WITH SAMPLES:
  avg() AS globalAvg
APPLY FILTER:
  AND(NOT(after("2019-08-29T10:00:00Z")))
USING EVENTS:
  AND(const(10.0, 5.0)) AS constantEvent,
  AND(lt(globalAvg)) FOR [20,] days AS low,
  AND(gt(globalAvg)) FOR [20,] days AS high
SELECT PERIODS:
  (constantEvent precedes (low follows high) WITHIN [0,2] days)
YIELD:
  all periods
```

Listing 6.14: DTSQL Query Expressing "UC14: N-Ary Event Sequences"

6.2 Performance Assessment

This section assesses the performance of the prototypical implementation of the language specification, as described in Chapter 5. First, Section 6.2.1 is concerned with how the duration of the query evaluation process changes with increasingly large input time series. Afterwards, Section 6.2.2 interprets and compares the results of the queries presented in Section 6.1 with what human intuition yields.

Both subsections operate on the same signal depicted in Figure 6.1. The concrete dates and values of its 743 data points are provided in Appendix A.3. For the quantitative evaluation, additional larger input data has been artificially created by gradually increasing the sampling rate, with linear interpolation between data points. That way, the number of data points was roughly doubled each time until the resulting time series had reached a size of about 3 million data points, equating to a CSV file of 110 MB.

The input data files as well as the source code of benchmark scripts utilized in this evaluation are available at `https://github.com/dtsql-oss/scripting`.

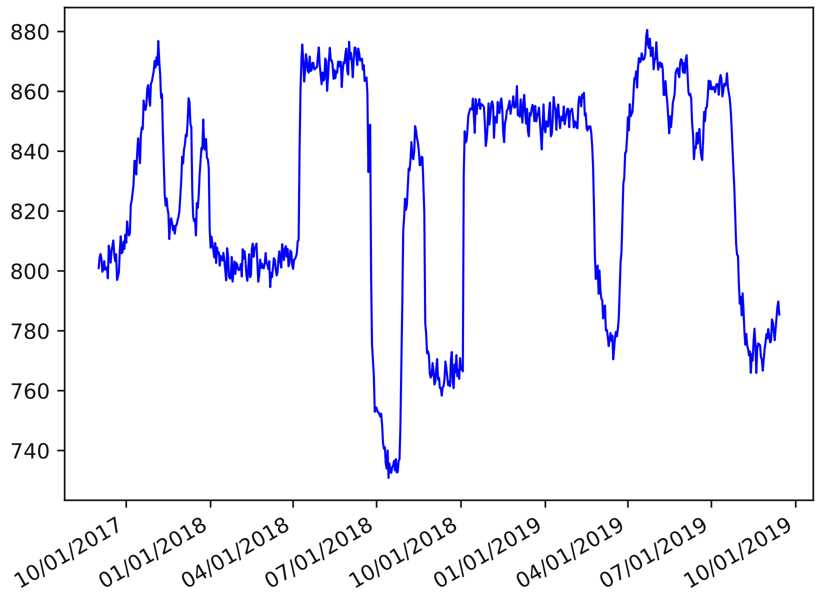


Figure 6.1: Input Time Series for Performance Assessment

6.2.1 Quantitative Analysis

The aim of this runtime performance analysis is *not* to provide an exhaustive, highly sophisticated benchmark. It should rather give a general indication of how the reference implementation's runtime changes with increasing input size. Furthermore, the measurements are only concerned with the amount of time elapsed during the query evaluation process. Other computational tasks such as parsing the input data or converting the query result into a format that is transmissible via the web service are not included because they are not strictly part of the system's core functionalities.

The benchmark consists of the fourteen use-case-specific queries presented in Section 6.1. It was conducted on a machine running Windows 11, an Intel Core i9-9900K CPU with 3.60 GHz and 32 GB of RAM. The queries were executed ten times on twelve time series, each describing the signal from Figure 6.1 with increasing levels of accuracy and size.

While we recognize that there are mathematical arguments for using the minimum of multiple performance measurements [70], we opted to take the arithmetic mean of the ten query executions per use case and input size. This is because, in practice, the system would not be running in an isolated environment and therefore, also be subject to noise. The performance assessment should give indications of what runtime performance is to be expected in general, not only under ideal conditions.

The results of the benchmarks are visualized in Figure 6.2. The horizontal axis contains the size of the input time series. The vertical axis is scaled logarithmically and represents the average query processing time for the given use case and input size. A table with the concrete measured values illustrated by this figure can be found in Appendix A.4.

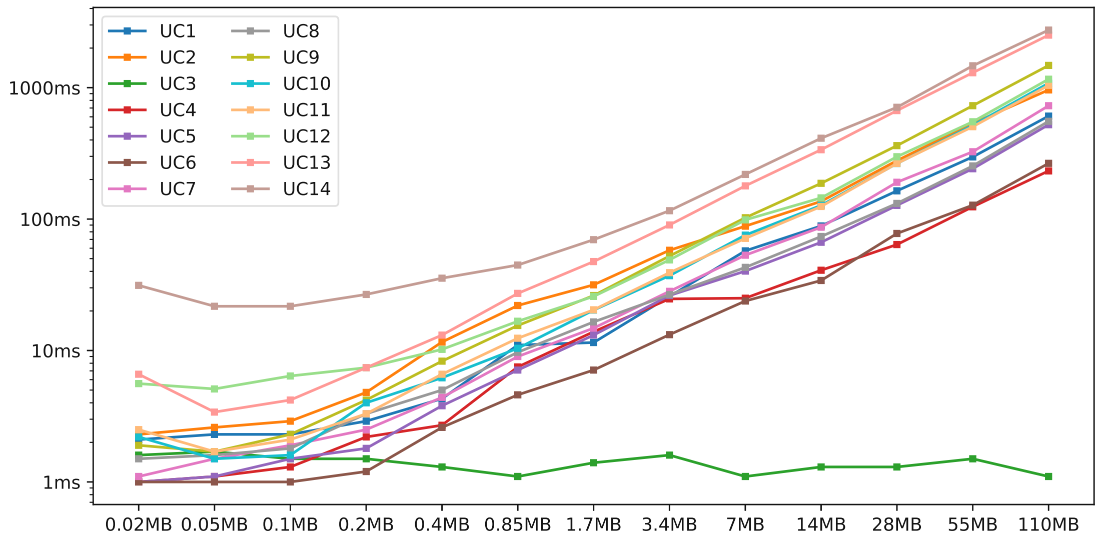


Figure 6.2: Progression of Query Evaluation Runtime in Relation to Input Size

At a wholistic glance, the diagram shows that it took around 2–3 seconds to evaluate the computationally most expensive queries on the largest input time series. Compared to the smaller input sizes—where evaluation times are (significantly) less than 100 milliseconds—, there are relatively substantial increases in runtime starting at around 7 MB. Profiling efforts conducted after the benchmark revealed that the reason for this are repeated scans of the whole time series. The algorithmic components for detecting and computing the various kinds of events and aggregates are, on their own, implemented efficiently. However, when combining them—as it often happens in the benchmark queries—, they all individually iterate over the input time series, which, in total, contributes noticeably to the observed runtime increases.

While this is evidently not ideal in theory, it is not a grave drawback in practice. The absolute runtime of the query evaluation is, with a few seconds, in a margin that is acceptable for the domain experts accompanying this thesis. Furthermore, the size of time series they process at once is well within the range of the ones in the benchmark data set. Nevertheless, the prototype should, of course, be improved in this regard in the future.

6.2.2 Qualitative Analysis

This subsection displays and assesses the results obtained by the prototypical DTSQL implementation from the queries shown in Section 6.1.

UC1: Global Aggregates

The result of the global aggregate calculation depicted in Listing 6.1 is presented in Table 6.1. One could carry out manual calculations—e.g., using a script—to verify that the values are correct.

aggregate	value
`myAverage`	826.9558
`myStdDev`	37.1425
`myMin`	730.8675
`myMax`	880.5267
`mySum`	613,601.1758
`myCount`	742.0000

Table 6.1: Result of Listing 6.1 ("UC1: Global Aggregates") Given Figure 6.1

UC2: Local Aggregates

The result of the local aggregate calculation depicted in Listing 6.2 is presented in Table 6.2. Again, manual calculations could verify that the values obtained by the reference implementation are correct.

aggregate	value
`localAverage`	827.2236
`localStdDev`	37.1609
`localMin`	730.8675
`localMax`	880.5267
`localSum`	600,130.5449
`localCount`	733.0000

Table 6.2: Result of Listing 6.2 ("UC2: Local Aggregates") Given Figure 6.1

UC3: Temporal Aggregates

The result of the temporal aggregate calculation depicted in Listing 6.3 is presented in Table 6.3. As before, the correctness of the values in the table could be confirmed using manual calculations.

UC4: Numerical Integral

The result of the remaining aggregate functions—global and local integrals—as depicted in Listing 6.4, is presented in Table 6.4. Similar to before, the correctness of these values may be verified using a simple script or manual calculations.

aggregate	value
`tempAverage`	486,240.0000
`tempStdDev`	279.2959
`tempMin`	6,652,800.0000
`tempMax`	17,400.0000
`tempSum`	144.7143
`tempCount`	3.0000

Table 6.3: Result of Listing 6.3 (“UC3: Temporal Aggregates”) Given Figure 6.1

aggregate	value
`globalIntegral`	52,946,454,865.1791
`localIntegral`	2,312,475,894.0000

Table 6.4: Result of Listing 6.4 (“UC4: Numerical Integral”) Given Figure 6.1

UC5: Threshold Filters

The result of the filter application depicted in Listing 6.5 is illustrated in Figure 6.3. In order to visualize the result more clearly, the included and excluded data points are shown in a scatter plot. That way, for instance, it is evident that the few data points with values greater than 875.5 have been filtered out correctly.

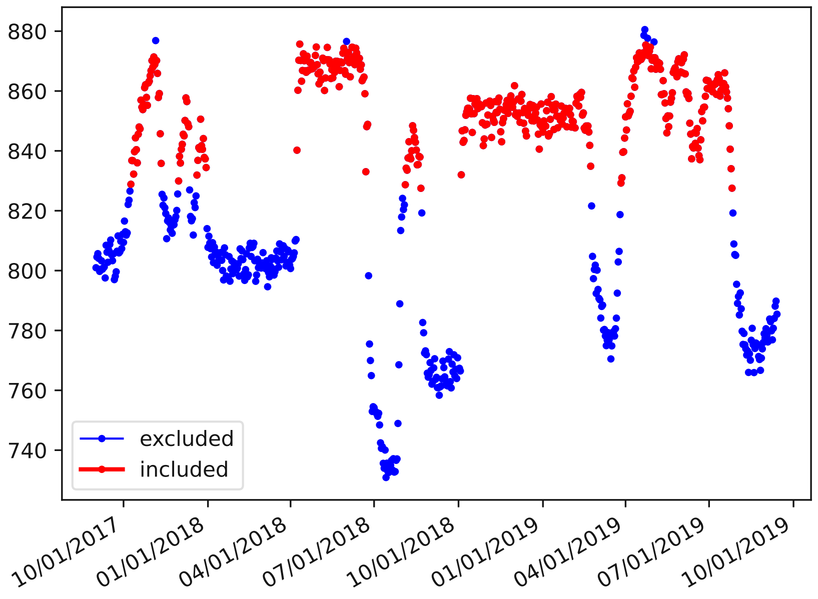


Figure 6.3: Result of Listing 6.5 (“UC5: Threshold Filters”) Given Figure 6.1

UC6: Temporal Filters

The result of the filter application depicted in Listing 6.6 is illustrated in Figure 6.4. It shows how the disjunctive temporal filter has been evaluated correctly to exclude the time range from `2018-01-13T10:00:00Z` (exclusive) until `2019-08-07T10:00:00Z` (inclusive).

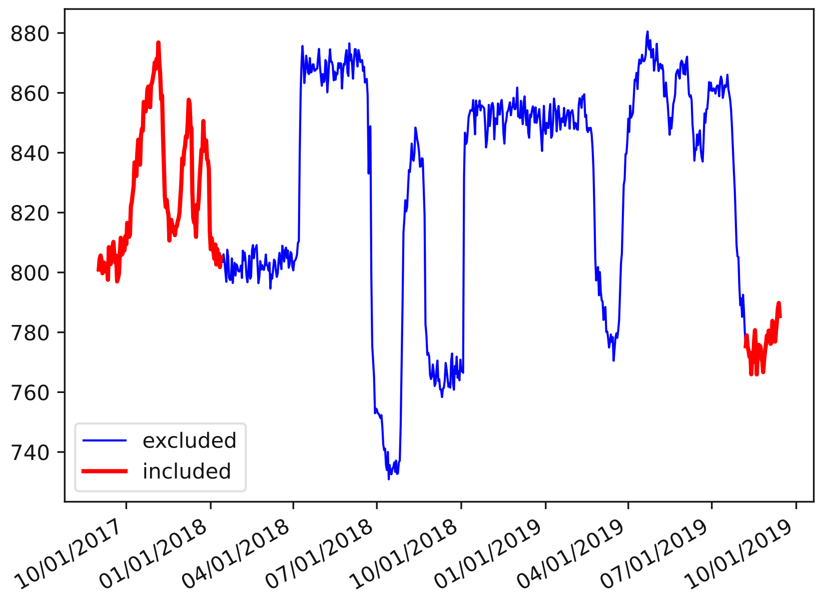


Figure 6.4: Result of Listing 6.6 (“UC6: Temporal Filters”) Given Figure 6.1

UC7: Threshold Events

The result of the threshold event query depicted in Listing 6.7 is shown in Figure 6.5. Note that the start and ending points of the red result intervals do not coincide exactly with the line representing the average because the time series consists of discrete values, i.e., there are time-gaps between the data points.

UC8: Deviation Events

The longest interval in the value range specified by the query depicted in Listing 6.8 is visualized in Figure 6.6. The acceptable range is illustrated using two horizontal, green lines. Even though it seems like there are intervals within this range that are longer than the highlighted one, that is not the case. Again, due to the discrete nature of the input time series, these intervals are in fact shorter than they appear in this continuous visualization (because the data points are closer to each other than they seem), or do not even contain a data point (e.g., during a very short, steep increase).

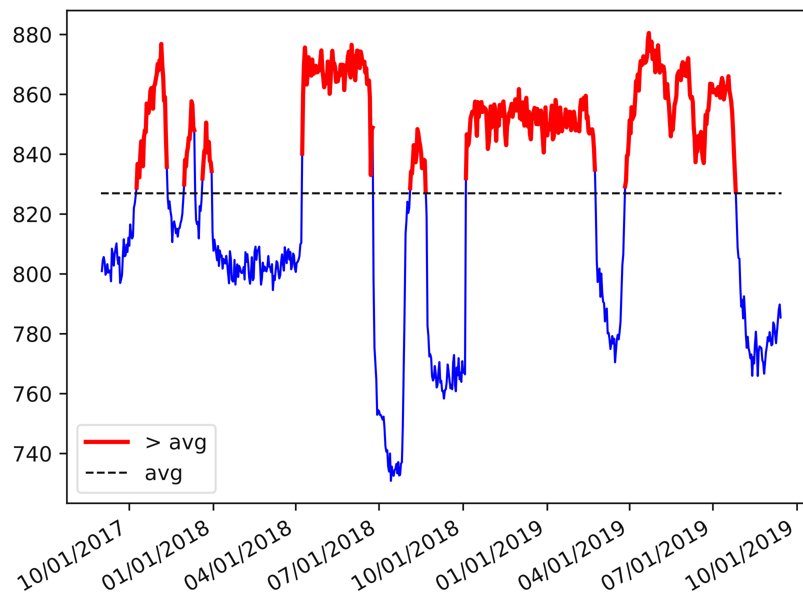


Figure 6.5: Result of Listing 6.7 ("UC7: Threshold Events") Given Figure 6.1

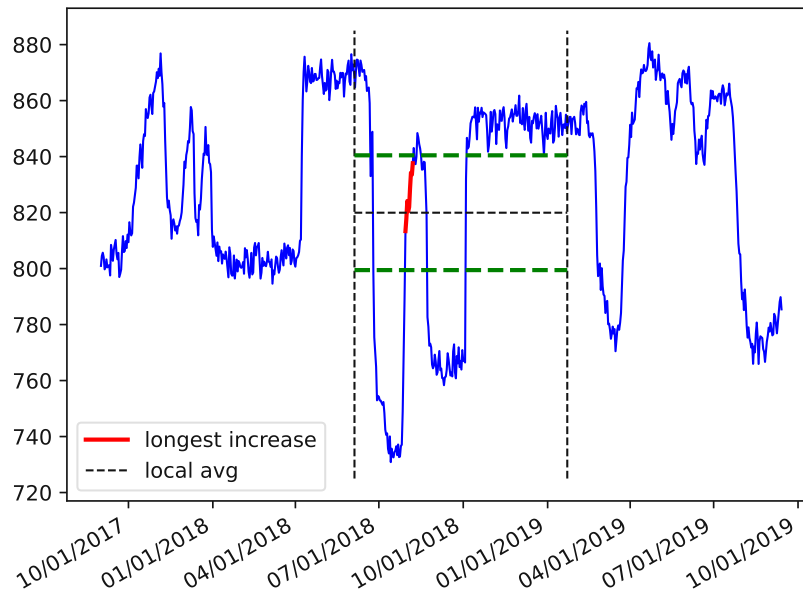


Figure 6.6: Result of Listing 6.8 ("UC8: Deviation Events") Given Figure 6.1

UC9: Constant Events

The constant periods detected by the system, as per the query in Listing 6.9, are depicted in Figure 6.7. The system has captured intervals which predominantly are in accord with what humans would classify as constant intervals. One might argue that the interval around `10/01/2018` should also be considered constant. Furthermore, the last detected interval, around `01/01/2019`, could also be narrowed down, i.e., the start could be a small amount of time later, and the end a small amount of time earlier.

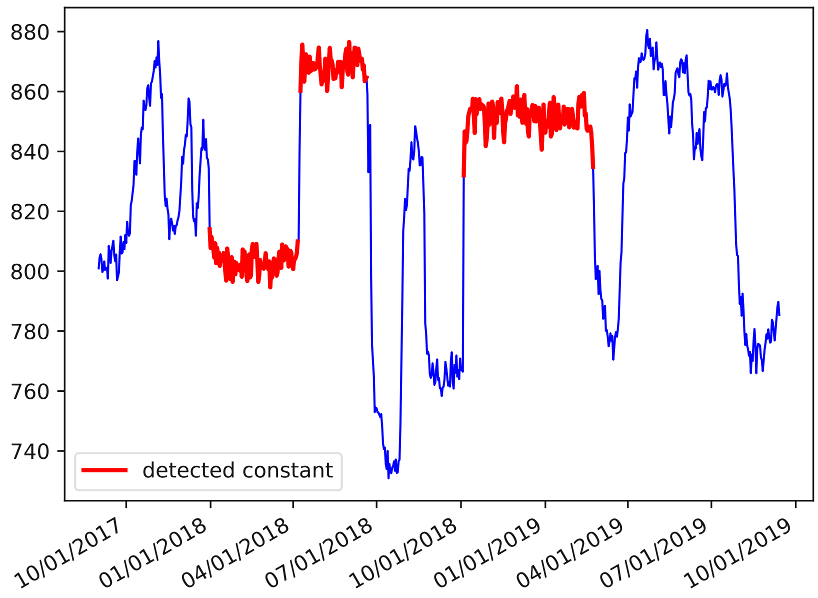


Figure 6.7: Result of Listing 6.9 (“UC9: Constant Events”) Given Figure 6.1

UC10: Monotonic Events

The intervals of increasing values which have been captured as a result of the query in Listing 6.10 are shown in Figure 6.8. Recall that this query explicitly only regards increases of around 5 % to 25 %. It does seem to be in accord with what humans might consider a moderate increase. However, there is one increase—directly after the second highlighted period—which was not detected. This could be amended with slightly different query parameters.

UC11: Duration Constraints for Events

The query in Listing 6.11 is nearly identical to the one from the previous use case. The only difference is that it is only concerned with intervals which are between 2 and 5 weeks long. Such a duration constraint might be useful to define whether a steep increase or decrease happens fast or slowly. Therefore, the result of the query—depicted in Figure 6.9—is equivalent to the one from before, except for three intervals which are either too long or too short.

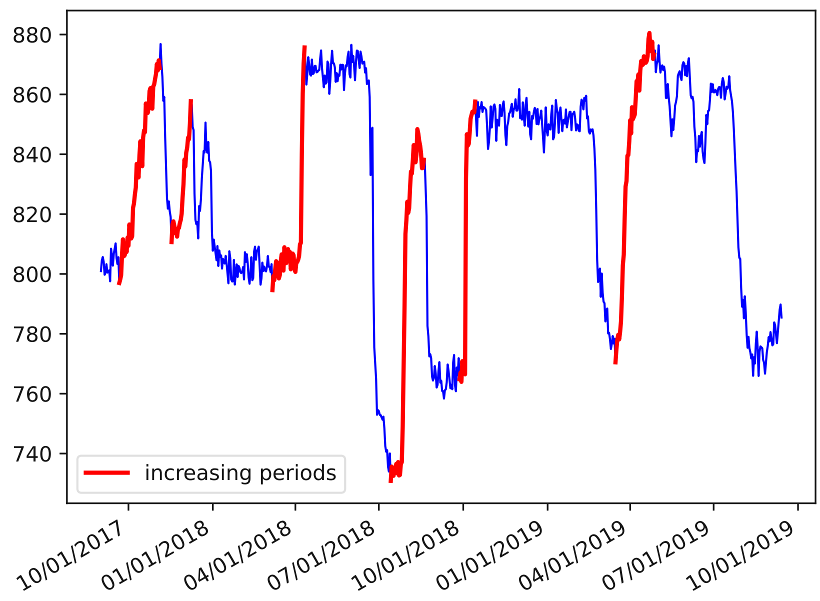


Figure 6.8: Result of Listing 6.10 (“UC10: Monotonic Events”) Given Figure 6.1

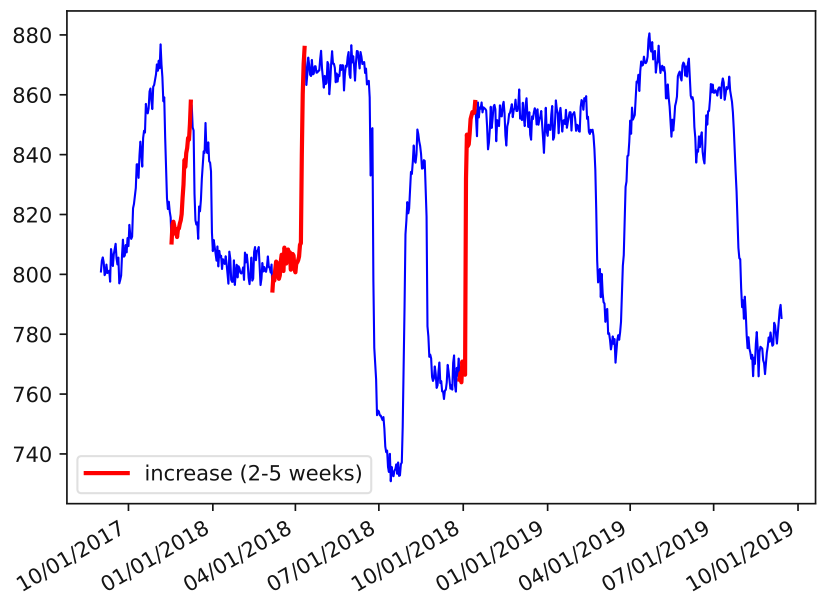


Figure 6.9: Result of Listing 6.11 (“UC11: Duration Constraints for Events”) Given Figure 6.1

UC12: Binary Event Sequences

Listing 6.12 represents a query that identifies intervals where a period with values above the global average appears before a period with values below the global average. Its result is visualized in Figure 6.10. Furthermore, recall that the query only considers detected intervals which have a minimum duration of 20 days. This is why seemingly matching periods—e.g., at the beginning of the time series—have not been detected. The detection of the event sequences align with what is visible to the human eye. Moreover, the automatic verification of the duration constraint(s) simplified filtering out non-relevant event sequences.

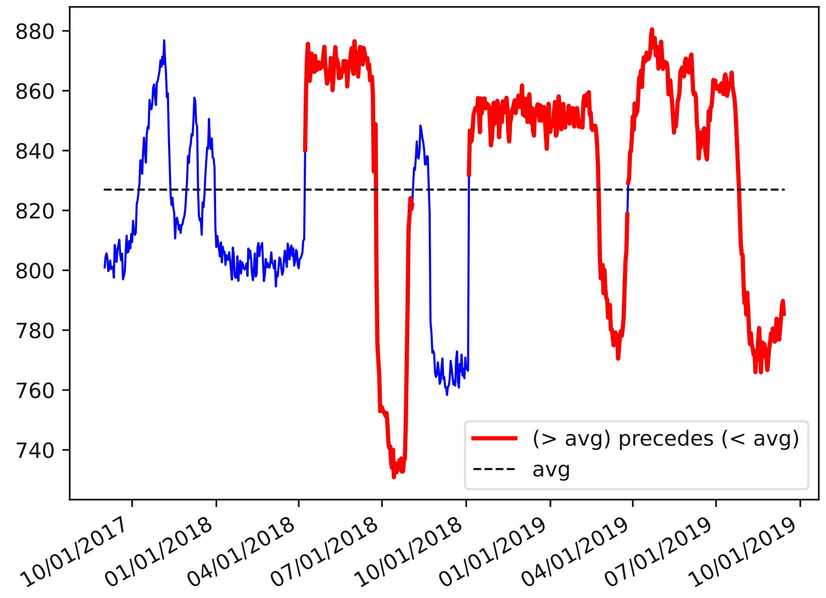


Figure 6.10: Result of Listing 6.12 ("UC12: Binary Event Sequences") Given Figure 6.1

UC13: Time Tolerance for Event Sequences

The result of the query depicted in Listing 6.13 is shown in Figure 6.11. The goal was to detect constant intervals which appear before increasing intervals, with a gap of 30 to 100 days between them. While the first captured period—the red one—seems very plausible, the second—the green one—probably is not what a human would have chosen. Intuitively, the interval should probably end with the increase which occurs around `07/01/2018`. However, since the reference implementation always tries to find maximal intervals satisfying the query constraints, it extended the interval until the next increase around `10/01/2018`. It would not have done so if it had considered the short period around `10/01/2018` as constant—see also remarks in "UC9: Constant Events". However, an improved version of the system should be able to refrain from maximizing such an event sequence, if it entails subsuming other notable events located in between (e.g., increases, decreases, constants).

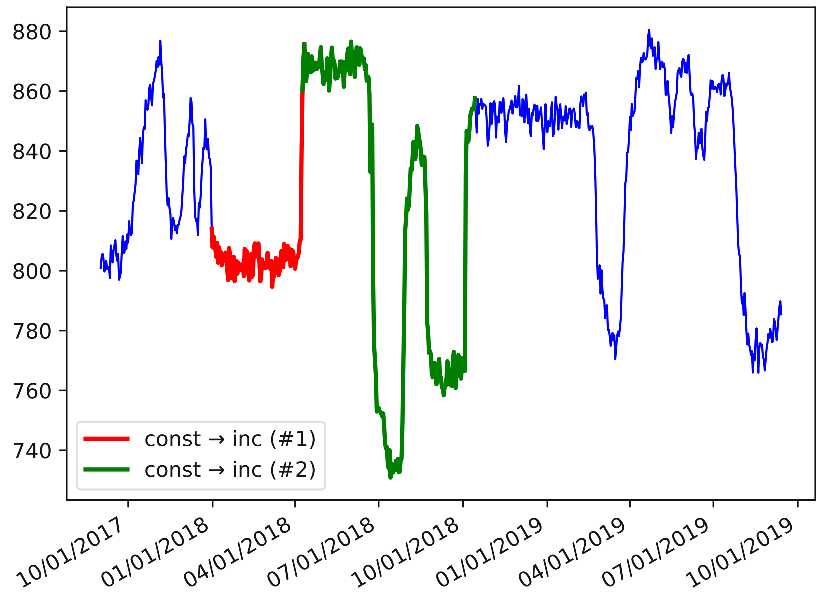


Figure 6.11: Result of Listing 6.13 ("UC13: Time Tolerance for Event Sequences") Given Figure 6.1

UC14: N-Ary Event Sequences

The result of the ternary event sequence represented by the query in Listing 6.14 is visualized in Figure 6.12. Again, the accuracy of the periods retrieved by the reference implementation depends on whether the period around `10/01/2018` should be considered constant or not. If the system had regarded it as a constant interval, then it would have returned two result periods (the second one spanning from slightly before `10/01/2018` until around `04/01/2019`).

6.3 Utility for Domain Experts

The previous subsections, especially Section 6.2.2, have shown that the query language specification and its manifestation in the form of the reference implementation perform reasonably well on the fourteen isolated use cases. However, a final discussion with the domain experts accompanying this thesis revealed aspects which are still missing for its contributions to be effectively applicable in practice.

The main factor deterring domain experts from using the developed system in their daily work is that it is not able to abstract away the mathematical thresholds and tolerance parameters that are required by event and filter definitions. While they are necessary for the system to verify the criteria defined in the language specification, it does not accurately model human behavior. Analysts rarely think about concrete threshold values

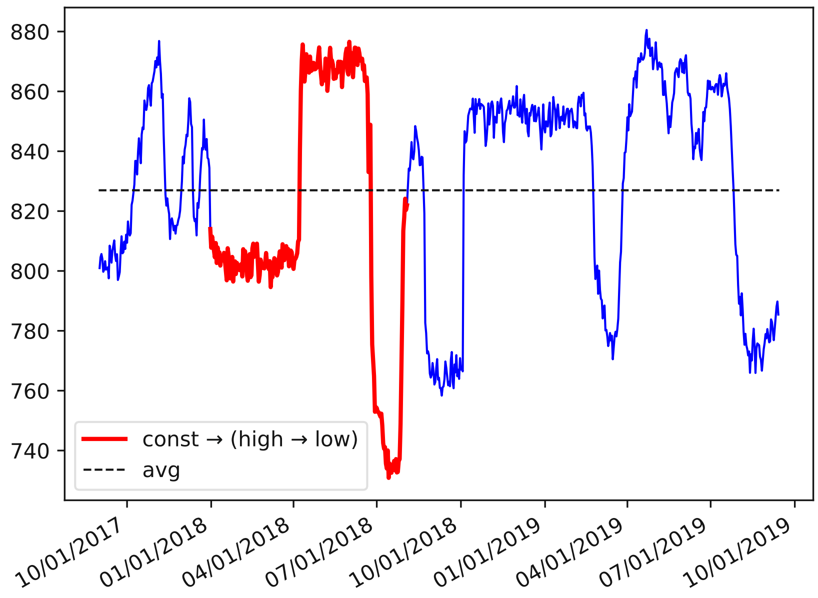


Figure 6.12: Result of Listing 6.14 ("UC14: N-Ary Event Sequences") Given Figure 6.1

and, for instance, never about slopes of regression lines or rates of instantaneous change when examining a time series. In our discussion, they maintained that these parameters are important and should be part of the language specification because they are a decent way to formally describe human intuition. Nevertheless, an additional abstraction layer on top of DTSQL which automatically determines sensible values for these parameters might be required.

Such a functionality is especially important since the parameters in question highly depend on the considered interval. For instance, locally significant peaks (increases) might not be discovered in two periods of the same time series with the same query parameters. This is due to differences in measures such as durations, value distributions or standard deviations. In order to detect the increases in both instances, the parameters need to be tweaked accordingly, based on the respectively examined period (or data instance in general, if we were talking about different time series). This burden, however, should not lie with the end user. Ideally, the aforementioned desired meta-layer on top of DTSQL should be able to infer appropriate parameter values based on the data currently focused in on as well as based on their temporal resolution.

A direct consequence of such a dynamic—i.e., data-aware—meta-layer would be that queries and their parameters would not have to be adjusted to newly ingested data. As explained above, the criteria whether an interval is noteworthy highly depends on aspects like its duration and fluctuations in the value dimension. These criteria change as the curve described by the time series changes. Subsequently, the query parameters need to be re-evaluated. This re-evaluation step could—and for scalability reasons, should—be automated by such a meta-layer, so as to avoid reusable, parameterized queries being rendered ineffective by new data.

Putting this shortcoming aside, the system, in its current form, does already provide benefits to the domain experts. The feature set of DTSQL allows them to express periods and occurrences of note in a manner they currently are not able to. The declarative nature of this DSL furthermore allows for a clear separation of conveying intentions and ways of achieving them. In other words, the algorithms and computational tasks associated with the various language features can be improved and extended completely independent from existing queries. This is not possible with ad hoc scripts and programs specifically created for individual tasks.

Finally, feature-wise, DTSQL is able to cover the majority of objectives immediately relevant to the domain experts. They specifically mentioned that the supported aggregate functions (the concept of *samples*) are very useful to them. One important potential extension they mentioned, however, is support for multivariate time series. This would enable them to relate detected phenomena of different variables—e.g., temperature and pressure of the same machine—to each other, which is a valuable use case for them. In addition, they also expressed interest in being able to weaken the condition of immediate event sequences such that no other contextually relevant period may appear in between (instead of no other data point). This has also been recognized in the evaluation of "UC13: Time Tolerance for Event Sequences". For example, a constant period that follows an increase within 15 minutes should—if this criterion is applied—only be detected as event sequence if there is no decrease in between.

CHAPTER 7

Conclusion

7.1 Summary

Factory operators have been recording sensor data of their production machines for years. While this results in huge amounts of time series data possibly harboring valuable optimization potentials, these data can only be exploited if there are efficient means of querying and extracting information from them in a target-oriented manner. There are purpose-built temporal databases which are able to ingest and manage large amounts of time series data efficiently. They also provide rudimentary query capabilities which are suitable for low-level use cases such as basic filtering or grouping. What available time series databases mostly lack, however, are high-level query instruments allowing domain experts to analyze historical temporal data with respect to phenomena that are directly relevant to their domain.

In this thesis, a declarative domain-specific language (DSL) which aims to alleviate this problem has been designed, formally specified and implemented: DTSQL (Declarative Time Series Query Language). Its feature set is a result of a requirement collection process which consisted of several consultations with external domain experts whose company specializes in optimizing the energy efficiency of business clients. As a result, DTSQL offers functionality such as declaring global and local aggregate values (e.g., arithmetic mean, standard deviation, numerical integral) which can be reused throughout a query. Furthermore, input time series may be filtered based on temporal or value thresholds as well as by specifying an acceptable range in which data point values should reside.

Most importantly, DTSQL detects intervals, or periods, in a time series which are specified declaratively as a combination of event functions expressed in propositional logic. For example, this allows query creators to find periods in which all data points were measured after a specific date and have value components which do not reside within a 5% interval of the local average between the start of the recordings and the aforementioned threshold date. The query language is also able to express conceptually more involved intervals such as ones displaying a monotonic increase or an (approximately) constant signal.

All of DTSQL's features have been formally modelled and specified precisely. This allowed for the development of a reference implementation that offers exactly those features. While efficiency was a factor that was taken into account, the prototype's primary concern is conformance to the formal language specification. Additionally, a client environment has been developed that provides an easy way to utilize the reference implementation. It offers guidance during the query formulation process and makes use of a projectional editor that is equipped with auto-complete, domain-specific error messages and context-aware suggestions (intentions). The client environment also lets users execute a query on a given time series once the formulation process is complete as well as visualizes its result.

Finally, an evaluation of the reference implementation centered around example queries targeting the use cases gathered during the requirement collection process was conducted. This evaluation pointed out potential performance optimizations and, more importantly, showed that the language specification captures the demands of time series analysts quite well. In a subsequent discussion, the domain experts accompanying this thesis highlighted strengths of DTSQL as well as opportunities for improvement from their perspective. Those and other possible future research efforts are presented in the section below.

7.2 Future Work

Although the evaluation has shown that both the language specification and its reference implementation are suitable to handle the identified use cases, they do exhibit certain limitations. These shortcomings can be roughly divided into three categories: issues pertaining to the language design, the reference implementation or aspects which would increase the general utility. They are discussed in the following paragraphs.

It is immediately evident that the lack of support for subqueries and chained queries is detrimental to DTSQL's expressiveness. Currently, in order for queries to be able to build on previous queries, an implementation-specific caching mechanism needs to be put in place that implicitly retains query results for future access if needed. Ideally, the language specification should make it possible to formulate the whole analysis of a time series in a single query (or, at least, conceptually related parts of it). Making the evaluation module aware of the entire analysis would also enable optimizations like internal query rewriting. Furthermore, the selection component could be made more powerful. Possible extensions include additional Allen relations (e.g., for detecting overlapping events or ones that occur during others) or even incorporating more complex operators from temporal logics such as LTL (linear temporal logic) or CTL (computation tree logic). Moreover, as already mentioned in the evaluation, it would be more reasonable to weaken the condition for events following/preceding each other immediately to there being no otherwise captured period between them (instead of no data points).

It has been stated before that the reference implementation's main objective has been to stay fully conform to the language specification. While this is true semantics-wise, there are syntactic restrictions for filter and event definitions. They only support one "root" connective—i.e., conjunction or disjunction—with filter and event functions, respectively, as literals, but no boolean subexpressions (nested propositional formulas). Although this feature is already able to cover many practical use cases in its limited form, the full syntactic capabilities of DTSQL should be implemented in the future. Furthermore, the evaluation showed that, for large input sizes, there is potential for performance optimizations. Many algorithms involved in the query evaluation process need only one scan of the whole input series, i.e., exhibit linear time complexity with respect to the input size. However, the reference implementation executes the majority of them sequentially, thus negatively affecting the overall runtime. Mitigation measures for this include adopting a computational approach that performs all calculations in a single scan or executing multiple algorithms in parallel. The individual algorithms could even be specifically designed with parallelism in mind: The input time series could be split into smaller batches which are then processed concurrently—without having to hold all the data in memory at once—and afterwards, their local results would be combined into an overall result.

Again, the evaluation has already mentioned that having a meta-layer on top of the query module provided by the reference implementation which dynamically determines query parameters (e.g., for event functions) would improve the system's utility significantly. Ideally, this would be done based on the respective input time series, the fluctuations in its value dimension as well as relative durations of the phenomena it captures. This separation of mathematical modelling (parameters as required by the semantics specification) from domain-specific constraints (the definition of certain events—e.g., significant increases—differs from project to project) would also preserve the generality of the system as determining the parameters internally could potentially result in a bias towards the energy efficiency domain. Additionally, a crucial future use case is the handling of multivariate or multiple time series. This introduces challenges such as divergent sampling rates—which need to be accounted for, e.g., using linear interpolation—and different or even incompatible physical units in the value dimension. Finally, a more extensive quantitative and qualitative evaluation to gauge both the language specification's and the reference implementation's practicability is desirable. More specifically, it should include input data with less clear manifestations of significant periods, enabling more interesting comparisons of the programmatically retrieved results with data interpretations made by humans. Furthermore, substantially larger amounts of input data should be considered in order to obtain a longer-term trend regarding the runtime required by the query evaluation process.

APPENDIX A

Appendix

A.1 Full DTSQL Grammar

A.1.1 Parser Rules

```
parser grammar DtsqlParser;

options
{
  tokenVocab = DtsqlLexer;
}

dtsqlQuery
  :  WHITESPACE?
       (samplesDeclaration WHITESPACE)?
       (filtersDeclaration WHITESPACE)?
       (eventsDeclaration WHITESPACE)?
       (selectDeclaration WHITESPACE)?
       yieldDeclaration WHITESPACE?
       EOF
  ;

samplesDeclaration
  :  SAMPLES_CLAUSE COLON WHITESPACE aggregatorsDeclarationStatement
  ;

eventsDeclaration
  :  EVENTS_CLAUSE COLON WHITESPACE eventsDeclarationStatement
  ;

eventsDeclarationStatement
  :  eventList
  ;

```

```
eventList
  :  events LIST_SEPARATOR eventDeclaration
  |  eventDeclaration
  ;

events
  :  eventDeclaration (LIST_SEPARATOR eventDeclaration)*
  ;

eventDeclaration
  :  eventConnective WHITESPACE? (durationSpecification WHITESPACE?)?
    ↪ identifierDeclaration
  ;

eventConnective
  :  CONNECTIVE_IDENTIFIER PARENTHESIS_OPEN WHITESPACE? eventFunctionList
    ↪ WHITESPACE? PARENTHESIS_CLOSE
  ;

eventFunctionList
  :  eventFunctions LIST_SEPARATOR eventFunctionDeclaration
  |  eventFunctionDeclaration
  ;

eventFunctions
  :  eventFunctionDeclaration (LIST_SEPARATOR eventFunctionDeclaration)*
  ;

eventFunctionDeclaration
  :  singlePointFilterDeclaration
  |  complexEventDeclaration
  ;

complexEventDeclaration
  :  complexEvent
  |  negatedComplexEvent
  ;

complexEvent
  :  constantEvent
  |  increaseEvent
  |  decreaseEvent
  ;

negatedComplexEvent
  :  CONNECTIVE_NOT PARENTHESIS_OPEN WHITESPACE? complexEvent WHITESPACE?
    ↪ PARENTHESIS_CLOSE
  ;

constantEvent
  :  EVENT_CONSTANT PARENTHESIS_OPEN WHITESPACE? slope=scalarArgument
    ↪ LIST_SEPARATOR deviation=scalarArgument WHITESPACE? PARENTHESIS_CLOSE
  ;
```

```

increaseEvent
  :  EVENT_INCREASE PARENTHESIS_OPEN WHITESPACE? minChange=scalarArgument
    ↪ LIST_SEPARATOR monotonicUpperBound LIST_SEPARATOR tolerance=
    ↪ scalarArgument WHITESPACE? PARENTHESIS_CLOSE
  ;

decreaseEvent
  :  EVENT_DECREASE PARENTHESIS_OPEN WHITESPACE? minChange=scalarArgument
    ↪ LIST_SEPARATOR monotonicUpperBound LIST_SEPARATOR tolerance=
    ↪ scalarArgument WHITESPACE? PARENTHESIS_CLOSE
  ;

monotonicUpperBound
  :  scalarArgument
  |  HYPHEN
  ;

durationSpecification
  : EVENT_DURATION WHITESPACE TIME_UNIT
  ;

selectDeclaration
  :  SELECT_CLAUSE COLON WHITESPACE temporalRelation
  ;

temporalRelation
  :  PARENTHESIS_OPEN op1=IDENTIFIER WHITESPACE TEMPORAL_RELATION WHITESPACE
    ↪ op2=IDENTIFIER WHITESPACE? timeToleranceSpecification?
    ↪ PARENTHESIS_CLOSE  #EventEvent
  |  PARENTHESIS_OPEN op1=IDENTIFIER WHITESPACE TEMPORAL_RELATION WHITESPACE
    ↪ op2=temporalRelation WHITESPACE? timeToleranceSpecification?
    ↪ PARENTHESIS_CLOSE  #EventRecursive
  |  PARENTHESIS_OPEN op1=temporalRelation WHITESPACE TEMPORAL_RELATION
    ↪ WHITESPACE op2=IDENTIFIER WHITESPACE? timeToleranceSpecification?
    ↪ PARENTHESIS_CLOSE  #RecursiveEvent
  |  PARENTHESIS_OPEN op1=temporalRelation WHITESPACE TEMPORAL_RELATION
    ↪ WHITESPACE op2=temporalRelation WHITESPACE? timeToleranceSpecification
    ↪ ? PARENTHESIS_CLOSE  #RecursiveRecursive
  ;

timeToleranceSpecification
  : TIME_TOLERANCE WHITESPACE TIME_UNIT
  ;

yieldDeclaration
  :  YIELD COLON WHITESPACE yieldType
  ;

yieldType
  :  YIELD_ALL_PERIODS
  |  YIELD_LONGEST_PERIOD
  |  YIELD_SHORTEST_PERIOD
  |  YIELD_DATA_POINTS
  |  YIELD_SAMPLE WHITESPACE IDENTIFIER
  |  YIELD_SAMPLE_SET WHITESPACE identifierList
  ;

filtersDeclaration
  :  FILTER_CLAUSE COLON WHITESPACE filterConnective
  ;

filterConnective
  :  CONNECTIVE_IDENTIFIER PARENTHESIS_OPEN WHITESPACE? singlePointFilterList
   ↪  WHITESPACE? PARENTHESIS_CLOSE
  ;

aggregatorsDeclarationStatement
  :  aggregatorList
  ;

aggregatorList
  :  aggregators LIST_SEPARATOR aggregatorDeclaration
  |  aggregatorDeclaration
  ;

aggregators
  :  aggregatorDeclaration (LIST_SEPARATOR aggregatorDeclaration)*
  ;

aggregatorDeclaration
  :  aggregatorFunctionDeclaration WHITESPACE identifierDeclaration
  ;

identifierList
  :  identifiers LIST_SEPARATOR IDENTIFIER
  |  IDENTIFIER
  ;

identifiers
  :  IDENTIFIER (LIST_SEPARATOR IDENTIFIER)*
  ;

aggregatorFunctionDeclaration
  :  valueAggregatorDeclaration
  |  temporalAggregatorDeclaration
  ;

valueAggregatorDeclaration
  :  VALUE_AGGREGATOR_FUNCTION PARENTHESIS_OPEN WHITESPACE? timeRange?
   ↪ WHITESPACE? PARENTHESIS_CLOSE
  ;

timeRange
  :  STRING_LITERAL LIST_SEPARATOR STRING_LITERAL
  ;
```

```

temporalAggregatorDeclaration
  :  TEMPORAL_AGGREGATOR_FUNCTION PARENTHESIS_OPEN WHITESPACE? TIME_UNIT
    ↪ LIST_SEPARATOR intervalList WHITESPACE? PARENTHESIS_CLOSE
  |  UNITLESS_TEMPORAL_AGGREGATOR_FUNCTION PARENTHESIS_OPEN WHITESPACE?
    ↪ intervalList WHITESPACE? PARENTHESIS_CLOSE
  ;

intervalList
  :  intervals LIST_SEPARATOR STRING_LITERAL
  |  STRING_LITERAL
  ;

intervals
  :  STRING_LITERAL (LIST_SEPARATOR STRING_LITERAL)*
  ;

identifierDeclaration
  :  AS WHITESPACE IDENTIFIER
  ;

singlePointFilterList
  :  singlePointFilters LIST_SEPARATOR singlePointFilterDeclaration
  |  singlePointFilterDeclaration
  ;

singlePointFilters
  :  singlePointFilterDeclaration (LIST_SEPARATOR
    ↪ singlePointFilterDeclaration)*
  ;

singlePointFilterDeclaration
  :  singlePointFilter
  |  negatedSinglePointFilter
  ;

singlePointFilter
  :  thresholdFilter
  |  temporalFilter
  |  deviationFilter
  ;

negatedSinglePointFilter
  :  CONNECTIVE_NOT PARENTHESIS_OPEN WHITESPACE? singlePointFilter WHITESPACE
    ↪ ? PARENTHESIS_CLOSE
  ;

temporalFilter
  :  TEMPORAL_FILTER_TYPE PARENTHESIS_OPEN WHITESPACE? STRING_LITERAL
    ↪ WHITESPACE? PARENTHESIS_CLOSE
  ;

thresholdFilter
```

```
    :   THRESHOLD_FILTER_TYPE PARENTHESIS_OPEN WHITESPACE? scalarArgument
      ↪ WHITESPACE? PARENTHESIS_CLOSE
    ;

scalarArgument
    :   NUMBER
    |   IDENTIFIER
    ;

deviationFilter
    :   DEVIATION_FILTER_TYPE PARENTHESIS_OPEN WHITESPACE?
      ↪ deviationFilterArguments WHITESPACE? PARENTHESIS_CLOSE
    ;

deviationFilterArguments
    :   AROUND_FILTER_TYPE LIST_SEPARATOR reference=scalarArgument
      ↪ LIST_SEPARATOR deviation=scalarArgument
    ;
```

Listing A.1: All ANTLR Parser Rules for DTSQL Queries

A.1.2 Lexer Rules

```
lexer grammar DtsqlLexer;

WHITESPACE   :   WHITESPACE_CHARACTER+   ;
fragment WHITESPACE_CHARACTER
    :  ' '   |  '\r'   |  '\n'   |  '\r\n'   |  '\t'   ;

SAMPLES_CLAUSE   :  'WITH SAMPLES'   ;
EVENTS_CLAUSE   :  'USING EVENTS'   ;
FILTER_CLAUSE   :  'APPLY FILTER'   ;
SELECT_CLAUSE   :  'SELECT PERIODS'   ;

YIELD   :  'YIELD'   ;
YIELD_ALL_PERIODS   :  'all periods'   ;
YIELD_LONGEST_PERIOD   :  'longest period'   ;
YIELD_SHORTEST_PERIOD   :  'shortest period'   ;
YIELD_DATA_POINTS   :  'data points'   ;
YIELD_SAMPLE   :  'sample'   ;
YIELD_SAMPLE_SET  :  'samples'   ;

CONNECTIVE_NOT   :  'NOT'   ;
CONNECTIVE_IDENTIFIER   :  'AND'   |  'OR'   ;

THRESHOLD_FILTER_TYPE
    :   'gt'
    |   'lt'
    ;

TEMPORAL_FILTER_TYPE
    :   'before'
    |   'after'
    ;
```

```

DEVIATION_FILTER_TYPE
  :  'around'
  ;

AROUND_FILTER_TYPE
  :  'rel'
  |  'abs'
  ;

EVENT_CONSTANT  :  'const'  ;
EVENT_INCREASE  :  'increase'  ;
EVENT_DECREASE  :  'decrease'  ;

HYPHEN  :  '-'  ;
PARENTHESIS_OPEN  :  '('  ;
PARENTHESIS_CLOSE  :  ')'  ;
COLON  :  ':'  ;
fragment COMMA  :  ','  ;
LIST_SEPARATOR  :  WHITESPACE? COMMA WHITESPACE?  ;

NUMBER  :  INT  |  FLOAT  ;

fragment DIGIT  :  [0-9]  ;
fragment SIGN  :  '-'?  ;

INT  :  SIGN? DIGIT+  ;

FLOAT  :  SIGN? DIGIT+ '.' DIGIT+  ;

AS  :  'AS'  ;

TEMPORAL_RELATION
  :  'precedes'
  |  'follows'
  ;

VALUE_AGGREGATOR_FUNCTION
  :  'avg'
  |  'max'
  |  'min'
  |  'sum'
  |  'count'
  |  'integral'
  |  'stddev'
  ;

TEMPORAL_AGGREGATOR_FUNCTION
  :  'avg_t'
  |  'max_t'
  |  'min_t'
  |  'sum_t'
  |  'stddev_t'
  ;

UNITLESS_TEMPORAL_AGGREGATOR_FUNCTION  :  'count_t'  ;

TIME_TOLERANCE  :  TIME_TOLERANCE_WITHIN WHITESPACE DURATION_RANGE_OPEN
    ↪ WHITESPACE? INT? LIST_SEPARATOR INT? DURATION_RANGE_CLOSE  ;
fragment TIME_TOLERANCE_WITHIN  :  'WITHIN'  ;
fragment TIME_TOLERANCE_OPEN  :  PARENTHESIS_OPEN  |  '['  ;
fragment TIME_TOLERANCE_CLOSE  :  PARENTHESIS_CLOSE  | ']'  ;

EVENT_DURATION  :  DURATION_FOR WHITESPACE DURATION_RANGE_OPEN WHITESPACE?
    ↪ INT? LIST_SEPARATOR INT? DURATION_RANGE_CLOSE  ;
fragment DURATION_FOR  :  'FOR'  ;
fragment DURATION_RANGE_OPEN  :  PARENTHESIS_OPEN  |  '['  ;
fragment DURATION_RANGE_CLOSE  :  PARENTHESIS_CLOSE  | ']'  ;

TIME_UNIT
  :  'weeks'
  |  'days'
  |  'hours'
  |  'minutes'
  |  'seconds'
  |  'millis'
  ;

STRING_LITERAL  :  '"' STRING_CHARACTERS? '"'  ;
fragment STRING_CHARACTERS  :  STRING_CHARACTER+  ;
fragment STRING_CHARACTER  :  ~["\\\r\n]  ;

IDENTIFIER
  :  IDENTIFIER_FIRST_CHARACTER IDENTIFIER_CHARACTER*
  ;

fragment IDENTIFIER_FIRST_CHARACTER  :  LETTER_CHARACTER  ;
fragment IDENTIFIER_CHARACTER  :  LETTER_CHARACTER  |  DIGIT_CHARACTER  ;

fragment DIGIT_CHARACTER  :  DIGIT  ;
fragment LETTER_CHARACTER  :  [A-Za-z]  ;
```

Listing A.2: All ANTLR Lexer Rules for DTSQL Queries

A.2 Exemplary Time Series

```
2022-12-15 01:21:48.000;37.0
2022-12-15 01:36:48.000;41.0
2022-12-15 01:51:48.000;45.0
2022-12-15 02:06:48.000;65.0
2022-12-15 02:21:48.000;45.0
2022-12-15 02:36:48.000;43.0
2022-12-15 02:51:48.000;220.0
2022-12-15 03:06:48.000;270.0
2022-12-15 03:21:48.000;222.0
2022-12-15 03:36:48.000;265.0
2022-12-15 03:51:48.000;106.0
2022-12-15 04:06:48.000;112.0
2022-12-15 04:21:48.000;114.0
2022-12-15 04:36:48.000;113.0
2022-12-15 04:51:48.000;253.0
2022-12-15 05:06:48.000;291.0
2022-12-15 05:21:48.000;310.0
2022-12-15 05:36:48.000;314.0
2022-12-15 05:51:48.000;335.0
2022-12-15 06:06:48.000;299.0
2022-12-15 06:21:48.000;295.0
2022-12-15 06:36:48.000;277.0
2022-12-15 06:51:48.000;260.0
2022-12-15 07:06:48.000;259.0
2022-12-15 07:21:48.000;262.0
2022-12-15 07:36:48.000;277.0
2022-12-15 07:51:48.000;293.0
2022-12-15 08:06:48.000;-69.0
2022-12-15 08:21:48.000;-75.0
2022-12-15 08:36:48.000;-77.0
2022-12-15 08:51:48.000;-75.0
2022-12-15 09:06:48.000;-70.0
2022-12-15 09:21:48.000;-73.0
```

Listing A.3: Exemplary Time Series for Section 5.3 (Client Environment)

A.3 Evaluation Time Series

Please note that, in order to take up less space, the time of day has been left out. Every data point has been measured at `00:00:00Z`. Furthermore, there are four data points per line, separated by the pipe symbol (`|`). In order to correctly parse the time series, one would need to replace pipe symbols with line breaks beforehand.

```
Time,Value
2017-09-01,800.90 | 2017-09-02,804.53 | 2017-09-03,805.66 | 2017-09-04,804.06
2017-09-05,799.72 | 2017-09-06,800.46 | 2017-09-07,803.27 | 2017-09-08,800.44
2017-09-09,800.87 | 2017-09-10,801.21 | 2017-09-11,797.54 | 2017-09-12,808.45
2017-09-13,806.14 | 2017-09-14,802.77 | 2017-09-15,806.75 | 2017-09-16,808.72
2017-09-17,810.22 | 2017-09-18,805.28 | 2017-09-19,803.30 | 2017-09-20,805.59
2017-09-21,796.98 | 2017-09-22,798.12 | 2017-09-23,799.60 | 2017-09-24,806.48
2017-09-25,811.58 | 2017-09-26,805.90 | 2017-09-27,806.28 | 2017-09-28,809.80
2017-09-29,807.34 | 2017-09-30,812.11 | 2017-10-01,809.42 | 2017-10-02,816.54
2017-10-03,812.97 | 2017-10-04,811.80 | 2017-10-05,812.59 | 2017-10-06,822.09
2017-10-07,823.52 | 2017-10-08,826.55 | 2017-10-09,828.84 | 2017-10-10,836.77
2017-10-11,836.75 | 2017-10-12,832.20 | 2017-10-13,839.73 | 2017-10-14,844.33
2017-10-15,840.40 | 2017-10-16,835.99 | 2017-10-17,845.56 | 2017-10-18,847.97
2017-10-19,847.36 | 2017-10-20,856.95 | 2017-10-21,854.61 | 2017-10-22,853.78
2017-10-23,855.49 | 2017-10-24,861.11 | 2017-10-25,862.10 | 2017-10-26,857.85
2017-10-27,855.21 | 2017-10-28,862.69 | 2017-10-29,863.27 | 2017-10-30,865.07
2017-10-31,866.80 | 2017-11-01,870.15 | 2017-11-02,868.02 | 2017-11-03,871.38
2017-11-04,868.84 | 2017-11-05,876.85 | 2017-11-06,870.16 | 2017-11-07,865.88
2017-11-08,857.83 | 2017-11-09,859.15 | 2017-11-10,845.71 | 2017-11-11,835.81
2017-11-12,825.47 | 2017-11-13,821.81 | 2017-11-14,824.26 | 2017-11-15,821.00
2017-11-16,819.04 | 2017-11-17,810.65 | 2017-11-18,816.62 | 2017-11-19,817.58
2017-11-20,816.06 | 2017-11-21,813.59 | 2017-11-22,815.08 | 2017-11-23,812.46
2017-11-24,815.17 | 2017-11-25,815.34 | 2017-11-26,816.98 | 2017-11-27,818.03
2017-11-28,820.10 | 2017-11-29,825.59 | 2017-11-30,829.94 | 2017-12-01,838.18
2017-12-02,835.93 | 2017-12-03,840.53 | 2017-12-04,842.19 | 2017-12-05,845.54
2017-12-06,845.01 | 2017-12-07,850.15 | 2017-12-08,857.70 | 2017-12-09,856.47
2017-12-10,849.09 | 2017-12-11,848.06 | 2017-12-12,826.94 | 2017-12-13,818.09
2017-12-14,816.63 | 2017-12-15,817.36 | 2017-12-16,811.87 | 2017-12-17,822.68
2017-12-18,821.06 | 2017-12-19,824.94 | 2017-12-20,831.91 | 2017-12-21,836.82
2017-12-22,841.09 | 2017-12-23,840.57 | 2017-12-24,850.58 | 2017-12-25,841.70
2017-12-26,840.51 | 2017-12-27,844.11 | 2017-12-28,837.80 | 2017-12-29,837.32
2017-12-30,834.42 | 2017-12-31,814.01 | 2018-01-01,807.76 | 2018-01-02,811.48
2018-01-03,809.19 | 2018-01-04,807.31 | 2018-01-05,804.56 | 2018-01-06,809.33
2018-01-07,802.66 | 2018-01-08,807.85 | 2018-01-09,805.75 | 2018-01-10,806.20
2018-01-11,801.78 | 2018-01-12,805.06 | 2018-01-13,803.93 | 2018-01-14,803.47
2018-01-15,806.05 | 2018-01-16,803.43 | 2018-01-17,800.03 | 2018-01-18,796.87
2018-01-19,807.59 | 2018-01-20,805.29 | 2018-01-21,798.16 | 2018-01-22,797.49
2018-01-23,797.54 | 2018-01-24,804.68 | 2018-01-25,796.45 | 2018-01-26,800.55
2018-01-27,803.20 | 2018-01-28,798.97 | 2018-01-29,802.71 | 2018-01-30,802.07
2018-01-31,800.54 | 2018-02-01,800.20 | 2018-02-02,800.82 | 2018-02-03,802.32
2018-02-04,798.15 | 2018-02-05,807.27 | 2018-02-06,804.01 | 2018-02-07,806.61
2018-02-08,803.65 | 2018-02-09,798.84 | 2018-02-10,796.72 | 2018-02-11,800.33
2018-02-12,805.61 | 2018-02-13,797.91 | 2018-02-14,798.58 | 2018-02-15,807.53
2018-02-16,809.13 | 2018-02-17,804.72 | 2018-02-18,807.72 | 2018-02-19,808.03
2018-02-20,809.12 | 2018-02-21,803.27 | 2018-02-22,796.42 | 2018-02-23,798.54
2018-02-24,803.79 | 2018-02-25,800.97 | 2018-02-26,802.45 | 2018-02-27,801.12
```

```
2018-02-28,800.85 | 2018-03-01,802.57 | 2018-03-02,806.00 | 2018-03-03,802.45
2018-03-04,802.27 | 2018-03-05,800.59 | 2018-03-06,803.28 | 2018-03-07,794.60
2018-03-08,799.34 | 2018-03-09,797.92 | 2018-03-10,801.49 | 2018-03-11,804.27
2018-03-12,803.75 | 2018-03-13,800.59 | 2018-03-14,798.50 | 2018-03-15,799.64
2018-03-16,804.18 | 2018-03-17,806.55 | 2018-03-18,803.25 | 2018-03-19,801.12
2018-03-20,808.93 | 2018-03-21,806.23 | 2018-03-22,803.61 | 2018-03-23,807.47
2018-03-24,808.30 | 2018-03-25,804.77 | 2018-03-26,807.36 | 2018-03-27,801.55
2018-03-28,802.73 | 2018-03-29,806.81 | 2018-03-30,806.26 | 2018-03-31,801.84
2018-04-01,800.68 | 2018-04-02,803.76 | 2018-04-03,804.00 | 2018-04-04,804.72
2018-04-05,805.93 | 2018-04-06,809.90 | 2018-04-07,810.37 | 2018-04-08,840.22
2018-04-09,860.25 | 2018-04-10,870.20 | 2018-04-11,875.67 | 2018-04-12,870.19
2018-04-13,863.27 | 2018-04-14,866.85 | 2018-04-15,872.42 | 2018-04-16,870.10
2018-04-17,866.74 | 2018-04-18,866.25 | 2018-04-19,871.64 | 2018-04-20,866.80
2018-04-21,867.02 | 2018-04-22,869.50 | 2018-04-23,869.53 | 2018-04-24,867.36
2018-04-25,867.46 | 2018-04-26,867.99 | 2018-04-27,868.09 | 2018-04-28,871.89
2018-04-29,874.69 | 2018-04-30,868.80 | 2018-05-01,865.72 | 2018-05-02,862.37
2018-05-03,866.30 | 2018-05-04,863.56 | 2018-05-05,863.96 | 2018-05-06,870.96
2018-05-07,870.15 | 2018-05-08,860.17 | 2018-05-09,864.35 | 2018-05-10,870.47
2018-05-11,874.56 | 2018-05-12,870.40 | 2018-05-13,870.38 | 2018-05-14,869.02
2018-05-15,864.17 | 2018-05-16,867.12 | 2018-05-17,864.39 | 2018-05-18,866.64
2018-05-19,866.23 | 2018-05-20,869.92 | 2018-05-21,868.29 | 2018-05-22,869.90
2018-05-23,869.81 | 2018-05-24,861.43 | 2018-05-25,866.80 | 2018-05-26,869.70
2018-05-27,869.38 | 2018-05-28,872.64 | 2018-05-29,874.28 | 2018-05-30,866.77
2018-05-31,865.62 | 2018-06-01,876.56 | 2018-06-02,870.83 | 2018-06-03,872.88
2018-06-04,869.49 | 2018-06-05,864.73 | 2018-06-06,870.52 | 2018-06-07,874.63
2018-06-08,874.52 | 2018-06-09,871.48 | 2018-06-10,868.80 | 2018-06-11,874.30
2018-06-12,872.71 | 2018-06-13,870.27 | 2018-06-14,870.80 | 2018-06-15,870.93
2018-06-16,867.20 | 2018-06-17,868.74 | 2018-06-18,863.44 | 2018-06-19,864.08
2018-06-20,864.66 | 2018-06-21,859.08 | 2018-06-22,833.03 | 2018-06-23,848.09
2018-06-24,848.85 | 2018-06-25,798.28 | 2018-06-26,775.46 | 2018-06-27,769.90
2018-06-28,764.91 | 2018-06-29,752.93 | 2018-06-30,754.51 | 2018-07-01,754.19
2018-07-02,752.90 | 2018-07-03,752.83 | 2018-07-04,752.07 | 2018-07-05,751.31
2018-07-06,752.32 | 2018-07-07,748.42 | 2018-07-08,742.45 | 2018-07-09,740.59
2018-07-10,741.13 | 2018-07-11,735.60 | 2018-07-12,733.96 | 2018-07-13,740.04
2018-07-14,730.86 | 2018-07-15,735.65 | 2018-07-16,732.91 | 2018-07-17,732.56
2018-07-18,734.45 | 2018-07-19,734.96 | 2018-07-20,736.53 | 2018-07-21,733.36
2018-07-22,737.14 | 2018-07-23,732.66 | 2018-07-24,732.84 | 2018-07-25,736.60
2018-07-26,737.08 | 2018-07-27,748.92 | 2018-07-28,768.53 | 2018-07-29,788.90
2018-07-30,813.38 | 2018-07-31,817.90 | 2018-08-01,824.13 | 2018-08-02,820.39
2018-08-03,821.96 | 2018-08-04,828.68 | 2018-08-05,834.23 | 2018-08-06,833.54
2018-08-07,837.61 | 2018-08-08,843.06 | 2018-08-09,838.00 | 2018-08-10,837.30
2018-08-11,840.02 | 2018-08-12,848.39 | 2018-08-13,846.86 | 2018-08-14,844.44
2018-08-15,842.89 | 2018-08-16,840.27 | 2018-08-17,835.33 | 2018-08-18,835.44
2018-08-19,838.16 | 2018-08-20,837.93 | 2018-08-21,827.50 | 2018-08-22,819.26
2018-08-23,782.60 | 2018-08-24,779.22 | 2018-08-25,772.44 | 2018-08-26,773.18
2018-08-27,771.90 | 2018-08-28,765.72 | 2018-08-29,764.38 | 2018-08-30,765.25
2018-08-31,769.24 | 2018-09-01,766.64 | 2018-09-02,762.05 | 2018-09-03,762.92
2018-09-04,767.49 | 2018-09-05,770.54 | 2018-09-06,763.68 | 2018-09-07,764.32
2018-09-08,760.89 | 2018-09-09,760.84 | 2018-09-10,758.34 | 2018-09-11,761.15
2018-09-12,761.43 | 2018-09-13,763.84 | 2018-09-14,769.76 | 2018-09-15,767.58
2018-09-16,764.49 | 2018-09-17,761.80 | 2018-09-18,762.96 | 2018-09-19,761.48
2018-09-20,770.42 | 2018-09-21,772.92 | 2018-09-22,762.89 | 2018-09-23,760.80
2018-09-24,768.75 | 2018-09-25,766.13 | 2018-09-26,771.86 | 2018-09-27,764.80
```

```
2018-09-28,767.02 | 2018-09-29,763.93 | 2018-09-30,770.86 | 2018-10-01,767.02
2018-10-02,767.35 | 2018-10-03,766.45 | 2018-10-04,832.01 | 2018-10-05,846.69
2018-10-06,842.87 | 2018-10-07,843.47 | 2018-10-08,847.03 | 2018-10-09,851.89
2018-10-10,853.06 | 2018-10-11,854.22 | 2018-10-12,854.23 | 2018-10-13,853.97
2018-10-14,857.58 | 2018-10-15,852.36 | 2018-10-16,846.15 | 2018-10-17,857.43
2018-10-18,852.43 | 2018-10-19,855.70 | 2018-10-20,855.34 | 2018-10-21,857.44
2018-10-22,854.11 | 2018-10-23,855.80 | 2018-10-24,855.50 | 2018-10-25,855.26
2018-10-26,854.78 | 2018-10-27,849.38 | 2018-10-28,841.76 | 2018-10-29,843.98
2018-10-30,848.63 | 2018-10-31,855.99 | 2018-11-01,848.92 | 2018-11-02,851.97
2018-11-03,855.96 | 2018-11-04,856.63 | 2018-11-05,855.18 | 2018-11-06,844.48
2018-11-07,851.53 | 2018-11-08,850.27 | 2018-11-09,849.53 | 2018-11-10,856.47
2018-11-11,852.87 | 2018-11-12,852.70 | 2018-11-13,856.30 | 2018-11-14,857.64
2018-11-15,853.90 | 2018-11-16,846.38 | 2018-11-17,843.03 | 2018-11-18,849.42
2018-11-19,850.79 | 2018-11-20,853.57 | 2018-11-21,853.98 | 2018-11-22,855.42
2018-11-23,856.85 | 2018-11-24,852.52 | 2018-11-25,854.92 | 2018-11-26,855.46
2018-11-27,858.36 | 2018-11-28,854.58 | 2018-11-29,854.60 | 2018-11-30,856.37
2018-12-01,861.77 | 2018-12-02,852.04 | 2018-12-03,852.67 | 2018-12-04,851.54
2018-12-05,857.34 | 2018-12-06,852.65 | 2018-12-07,849.54 | 2018-12-08,856.60
2018-12-09,858.84 | 2018-12-10,852.79 | 2018-12-11,849.23 | 2018-12-12,854.17
2018-12-13,846.63 | 2018-12-14,844.99 | 2018-12-15,852.77 | 2018-12-16,851.88
2018-12-17,854.17 | 2018-12-18,855.49 | 2018-12-19,854.40 | 2018-12-20,850.06
2018-12-21,855.15 | 2018-12-22,849.99 | 2018-12-23,851.74 | 2018-12-24,852.62
2018-12-25,854.58 | 2018-12-26,850.31 | 2018-12-27,845.58 | 2018-12-28,840.58
2018-12-29,848.68 | 2018-12-30,855.66 | 2018-12-31,847.89 | 2019-01-01,846.28
2019-01-02,850.00 | 2019-01-03,848.45 | 2019-01-04,848.25 | 2019-01-05,853.41
2019-01-06,856.25 | 2019-01-07,844.98 | 2019-01-08,845.74 | 2019-01-09,850.50
2019-01-10,858.12 | 2019-01-11,855.26 | 2019-01-12,853.73 | 2019-01-13,847.13
2019-01-14,853.62 | 2019-01-15,855.62 | 2019-01-16,851.61 | 2019-01-17,847.69
2019-01-18,851.36 | 2019-01-19,851.54 | 2019-01-20,850.20 | 2019-01-21,854.97
2019-01-22,848.96 | 2019-01-23,851.00 | 2019-01-24,852.61 | 2019-01-25,854.16
2019-01-26,854.16 | 2019-01-27,848.03 | 2019-01-28,853.39 | 2019-01-29,853.62
2019-01-30,854.20 | 2019-01-31,852.04 | 2019-02-01,847.95 | 2019-02-02,849.98
2019-02-03,849.75 | 2019-02-04,847.85 | 2019-02-05,847.67 | 2019-02-06,855.90
2019-02-07,858.20 | 2019-02-08,857.51 | 2019-02-09,854.99 | 2019-02-10,858.87
2019-02-11,857.90 | 2019-02-12,859.54 | 2019-02-13,851.96 | 2019-02-14,852.60
2019-02-15,847.60 | 2019-02-16,846.90 | 2019-02-17,848.30 | 2019-02-18,847.60
2019-02-19,848.32 | 2019-02-20,846.29 | 2019-02-21,841.83 | 2019-02-22,834.87
2019-02-23,821.60 | 2019-02-24,804.75 | 2019-02-25,797.30 | 2019-02-26,800.36
2019-02-27,801.80 | 2019-02-28,792.36 | 2019-03-01,800.14 | 2019-03-02,793.69
2019-03-03,790.71 | 2019-03-04,790.24 | 2019-03-05,784.14 | 2019-03-06,788.02
2019-03-07,788.45 | 2019-03-08,780.11 | 2019-03-09,780.32 | 2019-03-10,778.16
2019-03-11,774.92 | 2019-03-12,776.76 | 2019-03-13,779.26 | 2019-03-14,776.75
2019-03-15,778.38 | 2019-03-16,770.48 | 2019-03-17,774.83 | 2019-03-18,778.13
2019-03-19,779.68 | 2019-03-20,778.13 | 2019-03-21,780.58 | 2019-03-22,784.11
2019-03-23,792.46 | 2019-03-24,802.88 | 2019-03-25,806.41 | 2019-03-26,818.70
2019-03-27,829.24 | 2019-03-28,831.00 | 2019-03-29,839.51 | 2019-03-30,839.71
2019-03-31,844.27 | 2019-04-01,851.35 | 2019-04-02,846.96 | 2019-04-03,855.71
2019-04-04,851.70 | 2019-04-05,852.49 | 2019-04-06,853.24 | 2019-04-07,858.25
2019-04-08,864.40 | 2019-04-09,864.12 | 2019-04-10,866.68 | 2019-04-11,861.26
2019-04-12,867.95 | 2019-04-13,871.13 | 2019-04-14,869.74 | 2019-04-15,869.97
2019-04-16,872.73 | 2019-04-17,870.77 | 2019-04-18,870.55 | 2019-04-19,871.09
2019-04-20,872.70 | 2019-04-21,878.58 | 2019-04-22,880.52 | 2019-04-23,875.39
2019-04-24,874.43 | 2019-04-25,877.60 | 2019-04-26,871.80 | 2019-04-27,873.31
```

```
2019-04-28,874.66 | 2019-04-29,867.40 | 2019-04-30,870.28 | 2019-05-01,871.11
2019-05-02,876.39 | 2019-05-03,871.12 | 2019-05-04,867.28 | 2019-05-05,869.24
2019-05-06,869.70 | 2019-05-07,868.11 | 2019-05-08,869.26 | 2019-05-09,867.26
2019-05-10,858.80 | 2019-05-11,858.83 | 2019-05-12,863.46 | 2019-05-13,859.08
2019-05-14,856.00 | 2019-05-15,850.81 | 2019-05-16,845.99 | 2019-05-17,851.67
2019-05-18,848.07 | 2019-05-19,851.64 | 2019-05-20,856.28 | 2019-05-21,857.56
2019-05-22,859.41 | 2019-05-23,866.19 | 2019-05-24,867.13 | 2019-05-25,867.74
2019-05-26,865.72 | 2019-05-27,864.76 | 2019-05-28,869.74 | 2019-05-29,870.77
2019-05-30,870.26 | 2019-05-31,866.37 | 2019-06-01,869.45 | 2019-06-02,866.08
2019-06-03,870.64 | 2019-06-04,872.09 | 2019-06-05,865.70 | 2019-06-06,859.69
2019-06-07,858.68 | 2019-06-08,859.20 | 2019-06-09,857.50 | 2019-06-10,849.25
2019-06-11,845.60 | 2019-06-12,837.34 | 2019-06-13,841.43 | 2019-06-14,840.96
2019-06-15,846.10 | 2019-06-16,842.51 | 2019-06-17,845.92 | 2019-06-18,847.45
2019-06-19,840.91 | 2019-06-20,838.42 | 2019-06-21,837.04 | 2019-06-22,843.69
2019-06-23,853.14 | 2019-06-24,850.00 | 2019-06-25,854.48 | 2019-06-26,854.62
2019-06-27,858.22 | 2019-06-28,863.60 | 2019-06-29,860.82 | 2019-06-30,863.41
2019-07-01,860.68 | 2019-07-02,860.70 | 2019-07-03,861.51 | 2019-07-04,861.01
2019-07-05,859.73 | 2019-07-06,861.93 | 2019-07-07,862.47 | 2019-07-08,862.35
2019-07-09,858.86 | 2019-07-10,863.76 | 2019-07-11,865.46 | 2019-07-12,864.11
2019-07-13,858.31 | 2019-07-14,860.18 | 2019-07-15,862.57 | 2019-07-16,861.87
2019-07-17,862.49 | 2019-07-18,866.06 | 2019-07-19,861.39 | 2019-07-20,859.65
2019-07-21,857.62 | 2019-07-22,854.08 | 2019-07-23,848.36 | 2019-07-24,840.56
2019-07-25,834.04 | 2019-07-26,827.53 | 2019-07-27,819.23 | 2019-07-28,808.89
2019-07-29,805.36 | 2019-07-30,805.09 | 2019-07-31,795.42 | 2019-08-01,789.04
2019-08-02,791.35 | 2019-08-03,785.16 | 2019-08-04,792.54 | 2019-08-05,787.22
2019-08-06,779.72 | 2019-08-07,775.33 | 2019-08-08,779.00 | 2019-08-09,775.20
2019-08-10,773.89 | 2019-08-11,771.77 | 2019-08-12,773.10 | 2019-08-13,765.97
2019-08-14,772.20 | 2019-08-15,770.05 | 2019-08-16,776.72 | 2019-08-17,780.71
2019-08-18,775.26 | 2019-08-19,765.92 | 2019-08-20,773.98 | 2019-08-21,775.81
2019-08-22,775.52 | 2019-08-23,775.20 | 2019-08-24,771.29 | 2019-08-25,770.29
2019-08-26,766.67 | 2019-08-27,770.77 | 2019-08-28,773.84 | 2019-08-29,775.95
2019-08-30,778.90 | 2019-08-31,777.49 | 2019-09-01,780.58 | 2019-09-02,778.38
2019-09-03,776.14 | 2019-09-04,776.49 | 2019-09-05,783.82 | 2019-09-06,782.94
2019-09-07,779.86 | 2019-09-08,776.87 | 2019-09-09,780.73 | 2019-09-10,784.01
2019-09-11,788.10 | 2019-09-12,789.79 | 2019-09-13,785.43
```

Listing A.4: Input Time Series for Chapter 6 (Evaluation)

A.4 Quantitative Evaluation Results

UC\MB	0.02	0.05	0.1	0.2	0.4	0.85	1.7	3.4	7	14	28	55	110
1	2.1	2.3	2.3	2.9	4.3	11.0	11.5	25.8	57.3	88.7	164.0	297.1	607.5
2	2.3	2.6	2.9	4.8	11.6	22.0	31.6	57.9	88.3	136.5	278.5	535.1	961.2
3	1.6	1.7	1.5	1.5	1.3	1.1	1.4	1.6	1.1	1.3	1.3	1.5	1.1
4	1.0	1.1	1.3	2.2	2.7	7.5	13.9	24.7	25.0	40.8	64.0	124.0	232.5
5	1.0	1.1	1.5	1.8	3.8	7.1	13.1	26.2	40.2	66.6	127.1	241.6	522.5
6	1.0	1.0	1.0	1.2	2.6	4.6	7.1	13.2	23.8	34.1	77.7	127.6	265.5
7	1.1	1.5	1.9	2.5	4.4	9.0	14.8	28.2	53.1	86.7	190.6	325.3	728.6
8	1.5	1.6	1.8	3.3	5.0	9.7	16.5	26.6	42.9	73.7	131.5	253.8	554.3
9	1.9	1.7	2.3	4.2	8.3	15.5	26.2	52.3	102.4	187.1	362.1	729.1	1475.9
10	2.2	1.5	1.6	4.0	6.2	10.4	20.3	37.1	75.5	127.2	264.6	517.1	1080.3
11	2.5	1.7	2.1	3.3	6.6	12.4	20.4	39.1	71.3	124.7	265.2	504.8	1041.2
12	5.6	5.1	6.4	7.4	10.2	16.7	25.8	48.9	98.6	145.0	298.2	550.1	1159.3
13	6.6	3.4	4.2	7.4	13.1	27.2	47.5	90.2	178.9	337.1	669.1	1298.8	2509.2
14	31.3	21.7	21.7	26.7	35.5	44.7	69.7	115.8	218.6	412.3	708.7	1466.7	2742.1

Table A.1: Progression of Query Evaluation Runtime in Relation to Input Size (Measurements in Milliseconds)

List of Figures

List of Tables

List of Listings

List of Algorithms

Glossary

ANTLR ANother Tool for Language Recognition; a parser generator used to process plain text DTSQL queries. 4, 76, 78, 80, 82, 84, 85, 89, 90, 96, 130, 132, 143

CSV Comma-Separated Value; used in files for information exchange. 87, 88, 100, 108

declarative programming paradigm which focuses on *what* problem should be solved instead of *how* this is achieved. xiii, 2, 10, 46, 51, 91, 120, 121

DSL Domain-Specific Language; programming language with limited expressiveness geared towards a particular problem domain; commonly declarative. 2, 9–12, 96, 100, 120, 121

DTSQL Declarative Time Series Query Language; the query language proposed by this thesis. xi, xiii, 2–5, 11, 12, 16, 19, 23, 25–31, 33–42, 45–61, 63–66, 68, 70, 72–85, 87–91, 94, 96–101, 103–108, 110, 119–123, 130, 132, 139, 143

JSON JavaScript Object Notation; lightweight format for data exchange; used in the DTSQL reference implementation. 89

language workbench development environments facilitating the creation of DSLs; often paired with projectional editors . 11, 12

MPS Meta Programming System; a language workbench and projectional editor developed by JetBrains. 11, 12, 89, 96, 97, 99–101, 139

projectional editor enables editing of documents based on a projection of their internal model (abstract syntax tree) instead of plain text. 2, 4, 5, 11, 12, 89, 96, 122

REST Representational State Transfer; a very common architecture for web services exchanges resources; used in the DTSQL reference implementation. 89

Spring Boot a Java framework for simplifying the creation of (web server) backends; used in the DTSQL reference implementation. 89

structure editor see projectional editor. 12

time series sequence of time-value pairs; commonly recorded by a sensor situated on a machine. xiii, 1, 7, 13

Bibliography

[1] C. M. Mastrangelo, J. R. Simpson, and D. C. Montgomery, “Time series analysis,” in *Encyclopedia of Operations Research and Management Science* (S. I. Gass and M. C. Fu, eds.), pp. 1546–1552, Boston, MA: Springer US, 2013.

[2] R. Adhikari and R. K. Agrawal, “An introductory study on time series modeling and forecasting,” *Computing Research Repository*, vol. abs/1302.6613, 2013.

[3] P. Esling and C. Agón, “Time-series data mining,” *ACM Computing Surveys*, vol. 45, no. 1, pp. 12:1–12:34, 2012.

[4] L. Deri, S. Mainardi, and F. Fusco, “tsdb: A compressed database for time series,” in *Traffic Monitoring and Analysis – 4th International Workshop, TMA 2012, Vienna, Austria, March 12, 2012. Proceedings* (A. Pescapè, L. Salgarelli, and X. A. Dimitropoulos, eds.), vol. 7189 of *Lecture Notes in Computer Science*, pp. 143–156, Springer, 2012.

[5] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, J. Teller, and K. Veeraraghavan, “Gorilla: A fast, scalable, in-memory time series database,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1816–1827, 2015.

[6] R. S. Weigel, D. M. Lindholm, A. Wilson, and J. Faden, “TSDS: high-performance merge, subset, and filter software for time series-like data,” *Earth Science Informatics*, vol. 3, no. 1-2, pp. 29–40, 2010.

[7] E. Dauterman, M. Rathee, R. A. Popa, and I. Stoica, “Waldo: A private time-series database from function secret sharing,” in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pp. 2450–2468, IEEE, 2022.

[8] Y. Hao, X. Qin, Y. Chen, Y. Li, X. Sun, Y. Tao, X. Zhang, and X. Du, “TS-Benchmark: A benchmark for time series databases,” in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pp. 588–599, IEEE, 2021.

[9] M. Jalal, S. Wehbi, S. Chilingaryan, and A. Kopmann, “SciTS: A benchmark for time-series databases in scientific experiments and industrial internet of things,” in

SSDBM 2022: 34th International Conference on Scientific and Statistical Database Management, Copenhagen, Denmark, July 6 – 8, 2022 (E. Pourabbas, Y. Zhou, Y. Li, and B. Yang, eds.), pp. 12:1–12:11, Association for Computing Machinery, 2022.

[10] P. Grzesik and D. Mrozek, "Comparative analysis of time series databases in the context of edge computing for low power sensor networks," in *Computational Science – ICCS 2020 – 20th International Conference, Amsterdam, The Netherlands, June 3-5, 2020, Proceedings, Part V* (V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, eds.), vol. 12141 of *Lecture Notes in Computer Science*, pp. 371–383, Springer, 2020.

[11] R. T. Snodgrass, I. Ahn, G. Ariav, D. S. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. G. Kulkarni, T. Y. C. Leung, N. A. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada, "TSQL2 language specification," *ACM Special Interest Group on Management of Data*, vol. 23, no. 1, pp. 65–86, 1994.

[12] R. T. Snodgrass, I. Ahn, G. Ariav, D. S. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. G. Kulkarni, T. Y. C. Leung, N. A. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada, "A TSQL2 tutorial," *ACM Special Interest Group on Management of Data*, vol. 23, no. 3, pp. 27–33, 1994.

[13] R. T. Snodgrass, ed., *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[14] M. H. Böhlen, J. Chomicki, R. T. Snodgrass, and D. Toman, "Querying TSQL2 databases with temporal logic," in *Advances in Database Technology – EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings* (P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, eds.), vol. 1057 of *Lecture Notes in Computer Science*, pp. 325–341, Springer, 1996.

[15] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings* (Y. Lakhnech and S. Yovine, eds.), vol. 3253 of *Lecture Notes in Computer Science*, pp. 152–166, Springer, 2004.

[16] V. Gutiérrez-Basulto and S. Klarman, "Towards a unifying approach to representing and querying temporal data in description logics," in *Web Reasoning and Rule Systems – 6th International Conference, RR 2012, Vienna, Austria, September 10-12, 2012. Proceedings* (M. Krötzsch and U. Straccia, eds.), vol. 7497 of *Lecture Notes in Computer Science*, pp. 90–105, Springer, 2012.

[17] V. Gutiérrez-Basulto, J. C. Jung, and A. Ozaki, "On metric temporal description logics," in *ECAI 2016 – 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands – Including Prestigious Applications of Artificial Intelligence (PAIS 2016)* (G. A. Kaminka, M. Fox, P. Bouquet, E. Hüllermeier, V. Dignum, F. Dignum, and F. van Harmelen, eds.), vol. 285 of *Frontiers in Artificial Intelligence and Applications*, pp. 837–845, IOS Press, 2016.

[18] S. Brandt, E. G. Kalayci, V. Ryzhikov, G. Xiao, and M. Zakharyaschev, "Querying log data with metric temporal logic," *Journal of Artificial Intelligence Research*, vol. 62, pp. 829–877, 2018.

[19] S. Brandt, D. Calvanese, E. G. Kalayci, R. Kontchakov, B. Mörzinger, V. Ryzhikov, G. Xiao, and M. Zakharyaschev, "Two-dimensional rule language for querying sensor log data: A framework and use cases," in *26th International Symposium on Temporal Representation and Reasoning, TIME 2019, October 16-19, 2019, Málaga, Spain* (J. Gamper, S. Pinchinat, and G. Sciavicco, eds.), vol. 147 of *LIPIcs*, pp. 7:1–7:15, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

[20] E. J. Keogh, H. Hochheiser, and B. Shneiderman, "An augmented visual query mechanism for finding patterns in time series data," in *Flexible Query Answering Systems, 5th International Conference, FQAS 2002, Copenhagen, Denmark, October 27-29, 2002, Proceedings* (T. Andreasen, A. Motro, H. Christiansen, and H. L. Larsen, eds.), vol. 2522 of *Lecture Notes in Computer Science*, pp. 240–250, Springer, 2002.

[21] K. Z. Haigh, W. Foslien, and V. Guralnik, "Visual query language: Finding patterns in and relationships among time series data," in *7th Workshop on Mining Scientific and Engineering Datasets, 24 April 2004, Lake Buena Vista, FL*, 05 2004.

[22] H. Hochheiser and B. Shneiderman, "Dynamic query tools for time series data sets: Timebox widgets for interactive exploration," *Information Visualization*, vol. 3, no. 1, pp. 1–18, 2004.

[23] M. Correll and M. Gleicher, "The semantics of sketch: Flexibility in visual query systems for time series data," in *11th IEEE Conference on Visual Analytics Science and Technology, IEEE VAST 2016, Baltimore, MD, USA, October 23-28, 2016* (G. L. Andrienko, S. Liu, and J. T. Stasko, eds.), pp. 131–140, IEEE Computer Society, 2016.

[24] K. Ryall, N. Lesh, T. Lanning, D. Leigh, H. Miyashita, and S. Makino, "Querylines: approximate query for visual browsing," in *Extended Abstracts Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2-7, 2005* (G. C. van der Veer and C. Gale, eds.), pp. 1765–1768, Association for Computing Machinery, 2005.

[25] A. Soylu, E. Kharlamov, D. Zheleznyakov, E. Jiménez-Ruiz, M. Giese, M. G. Skjæveland, D. Hovland, R. Schlatte, S. Brandt, H. Lie, and I. Horrocks, "OptiqueVQS:

A visual query system over ontologies for industry," *Semantic Web*, vol. 9, no. 5, pp. 627–660, 2018.

[26] D. H. Chau, C. Faloutsos, H. Tong, J. I. Hong, B. Gallagher, and T. Eliassi-Rad, "GRAPHITE: A visual query system for large graphs," in *Workshops Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy*, pp. 963–966, IEEE Computer Society, 2008.

[27] X. Ge and P. Smyth, "Deformable markov model templates for time-series pattern matching," in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, (New York, NY, USA), pp. 81–90, Association for Computing Machinery, 2000.

[28] J. Lin, "Finding motifs in time series," in *Proceedings of Special Interest Group on Knowledge Discovery and Data Mining, 2002, Edmonton, Alberta, Canada, July 23-26*, 07 2002.

[29] E. Keogh, "A fast and robust method for pattern matching in time series databases," *Proceedings of WUSS*, vol. 97, no. 1, p. 99, 1997.

[30] E. J. Keogh and P. Smyth, "A probabilistic approach to fast pattern matching in time series databases," in *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97), Newport Beach, California, USA, August 14-17, 1997* (D. Heckerman, H. Mannila, and D. Pregibon, eds.), pp. 24–30, AAAI Press, 1997.

[31] M. Gregory and B. Shneiderman, "Shape identification in temporal data sets," in *Expanding the Frontiers of Visual Analytics and Visualization* (J. Dill, R. A. Earnshaw, D. J. Kasik, J. A. Vince, and P. C. Wong, eds.), pp. 305–321, Springer, 2012.

[32] Y. Zhou, H. Ren, Z. Li, and W. Pedrycz, "An anomaly detection framework for time series data: An interval-based approach," *Knowlegde-Based Systems*, vol. 228, p. 107153, 2021.

[33] F. Liu, X. Zhou, J. Cao, Z. Wang, T. Wang, H. Wang, and Y. Zhang, "Anomaly detection in quasi-periodic time series based on automatic data segmentation and attentional LSTM-CNN," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 6, pp. 2626–2640, 2022.

[34] H. Ren, Z. Ye, and Z. Li, "Anomaly detection based on a dynamic markov model," *Information Sciences*, vol. 411, pp. 52–65, 2017.

[35] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[36] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Special Interest Group on Programming Languages*, vol. 35, no. 6, pp. 26–36, 2000.

[37] M. Fowler, *Domain-Specific Languages.* The Addison-Wesley signature series, Addison-Wesley, 2011.

[38] M. Ward, "Language oriented programming," *Software-Concepts and Tools*, vol. 15, pp. 147–161, 01 1994.

[39] S. Dmitriev, "Language oriented programming: The next programming paradigm," 2004.

[40] M. Voelter, "Language and IDE modularization and composition with MPS," in *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers* (R. Lämmel, J. Saraiva, and J. Visser, eds.), vol. 7680 of *Lecture Notes in Computer Science*, pp. 383–430, Springer, 2011.

[41] M. Voelter and V. Pech, "Language modularity with the MPS language workbench," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (M. Glinz, G. C. Murphy, and M. Pezzè, eds.), pp. 1449–1450, IEEE Computer Society, 2012.

[42] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Companion to the 25th Annual ACM Special Interest Group On Programming Languages Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, part of SPLASH 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA* (W. R. Cook, S. Clarke, and M. C. Rinard, eds.), pp. 307–309, Association for Computing Machinery, 2010.

[43] S. Kelly, K. Lyytinen, and M. Rossi, "MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment," in *Advanced Information System Engineering, 8th International Conference, CAiSE'96, Heraklion, Crete, Greece, May 20-24, 1996, Proceedings* (P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, eds.), vol. 1080 of *Lecture Notes in Computer Science*, pp. 1–21, Springer, 1996.

[44] L. C. L. Kats and E. Visser, "The Spoofax language workbench: rules for declarative specification of languages and ides," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA* (W. R. Cook, S. Clarke, and M. C. Rinard, eds.), pp. 444–463, Association for Computing Machinery, 2010.

[45] D. H. Lorenz and B. Rosenan, "Cedalion: a language for language oriented programming," in *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*

2011, part of SPLASH 2011, Portland, OR, USA, October 22 – 27, 2011 (C. V. Lopes and K. Fisher, eds.), pp. 733–752, Association for Computing Machinery, 2011.

[46] A. R. da Silva, S. Vlajic, S. Lazarevic, I. Antovic, V. Stanojevic, and M. Milic, "Preliminary experience using jetbrains MPS to implement a requirements specification language," in *9th International Conference on the Quality of Information and Communications Technology, QUATIC 2014, Guimaraes, Portugal, September 23-26, 2014*, pp. 134–137, IEEE Computer Society, 2014.

[47] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning, "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," *Computer Languages, Systems & Structures*, vol. 44, pp. 24–47, 2015.

[48] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning, "The state of the art in language workbenches – conclusions from the language workbench challenge," in *Software Language Engineering – 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings* (M. Erwig, R. F. Paige, and E. V. Wyk, eds.), vol. 8225 of *Lecture Notes in Computer Science*, pp. 197–217, Springer, 2013.

[49] SPLASH, "LWC@SLE 2016 – Language Workbench Challenge." `https://2016.splashcon.org/track/lwc2016`, 2016. Accessed on 2022-09-07.

[50] A. Gomolka and B. Humm, "Structure editors: Old hat or future vision?," in *Evaluation of Novel Approaches to Software Engineering – 6th International Conference, ENASE 2011, Beijing, China, June 8-11, 2011. Revised Selected Papers* (L. A. Maciaszek and K. Zhang, eds.), vol. 275 of *Communications in Computer and Information Science*, pp. 82–97, Springer, 2011.

[51] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: a controlled experiment," in *Proceedings of the 24th ACM Special Interest Group on Software Engineering International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016* (T. Zimmermann, J. Cleland-Huang, and Z. Su, eds.), pp. 763–774, Association for Computing Machinery, 2016.

[52] A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio, eds., *Domain-Specific Languages in Practice – with JetBrains MPS*. Springer, 2021.

[53] A. Wasilewska, *Logics for Computer Science – Classical and Non-Classical*. Springer, 1 ed., 2018.

[54] P. R. Turner, T. Arildsen, and K. Kavanagh, *Applied Scientific Computing – With Python*. Texts in Computer Science, Springer, 1 ed., 2018.

[55] M. C. Potter, J. L. Lessing, and E. F. Aboufadel, *Advanced Engineering Mathematics*. Springer, 4 ed., 2019.

[56] M. Oberguggenberger and A. Ostermann, *Analysis for Computer Scientists – Foundations, Methods, and Algorithms, Second Edition*. Undergraduate Topics in Computer Science, Springer, 2 ed., 2018.

[57] T. Parr, *The Definitive ANTLR 4 Reference*. Dallas, Texas and Raleigh, North Carolina: The Pragmatic Programmers, 2 ed., 2014.

[58] International Organization for Standardization, *Date and time – Representations for information exchange – Part 1: Basic rules*. ISO 8601-1:2019(E). Vernier, Geneva, Switzerland: International Organization for Standardization, 2019. URL: `https://www.iso.org/standard/70907.html`.

[59] International Organization for Standardization, *Date and time – Representations for information exchange – Part 2: Extensions*. ISO 8601-2:2019(E). Vernier, Geneva, Switzerland: International Organization for Standardization, 2019. URL: `https://www.iso.org/standard/70908.html`.

[60] B. P. Welford, “Note on a method for calculating corrected sums of squares and products,” *Technometrics*, vol. 4, pp. 419–420, Aug. 1962.

[61] T. F. Chan, G. H. Golub, and R. J. Leveque, “Algorithms for computing the sample variance: Analysis and recommendations,” *The American Statistician*, vol. 37, pp. 242–247, Aug. 1983.

[62] R. F. Ling, “Comparison of several algorithms for computing sample means and variances,” *Journal of the American Statistical Association*, vol. 69, pp. 859–866, Dec. 1974.

[63] A. Neumaier, “Rundungsfehleranalyse einiger verfahren zur summation endlicher summen,” *Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 54, no. 1, pp. 39–51, 1974.

[64] W. Kahan, “Further remarks on reducing truncation errors,” *Communications of the ACM*, vol. 8, p. 40, Jan. 1965.

[65] I. Babuska, “Numerical stability in mathematical analysis,” in *Information Processing, Proceedings of IFIP Congress 1968, Edinburgh, UK, 5–10 August 1968, Volume 1 – Mathematics, Software* (A. J. H. Morrel, ed.), pp. 11–23, 1968.

[66] X. Zhou, F. Wang, and C. Zaniolo, “Efficient temporal coalescing query support in relational database systems,” in *Database and Expert Systems Applications,*

17th International Conference, DEXA 2006, Kraków, Poland, September 4-8, 2006, Proceedings (S. Bressan, J. Küng, and R. R. Wagner, eds.), vol. 4080 of *Lecture Notes in Computer Science*, pp. 676–686, Springer, 2006.

[67] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo, "Coalescing in temporal databases," in *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India* (T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds.), pp. 180–191, Morgan Kaufmann, 1996.

[68] F. Campagne, *The MPS language workbench*, vol. I. North Charleston, South Carolina, USA: CreateSpace Independent Publishing Platform, 3 ed., Mar. 2016.

[69] F. Campagne, *The MPS language workbench*, vol. II. North Charleston, South Carolina, USA: CreateSpace Independent Publishing Platform, 1 ed., May 2016.

[70] J. Chen and J. Revels, "Robust benchmarking in noisy environments," *Computing Research Repository*, vol. abs/1608.04295, 2016.