TU WIEN Informatics

Temporal Reasoning in Knowledge Graphs

Artificial Intelligence Systems for Reasoning with Time in Vadalog

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Markus Nissl
Registration Number 01525567

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Emanuel Sallinger

The dissertation has been reviewed by:

Paolo Atzeni	Andrea Calì

Vienna, 3rd July, 2023

Markus Nissl

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

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Markus Nissl

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, 3. Juli 2023

Markus Nissl

Acknowledgements

A dissertation is the culmination of a protracted educational career. I was able to meet a lot of people over this period, engage in a lot of stimulating conversations, work on exciting projects and take a lot with me. I want to express my gratitude to everyone who supported me along the way.

I would especially like to acknowledge my supervisor, Emanuel Sallinger, who provided me with excellent guidance as I worked on my dissertation and gave me the chance to work in this fascinating field.

I also want to thank my parents, Michaela and Milan, for their continued support throughout the years and for enabling me to dedicate my complete attention to my studies, Clara for her support in stressful situations, and my dogs for simply listening when I was worried.

I also want to thank the online tools Google Translate and Quillbot for assisting with several English formulations, thereby helping to improve the readability of the thesis as well as the funding agency Vienna Science and Technology Fund (WWTF) for funding our project through grant VRG18-013.

Kurzfassung

Die zunehmende Bedeutung von Wissensgraphen (Knowledge Graphs) hat großes Interesse an der Bereitstellung skalierbarer und effizienter Reasoning (Schlussfolgerungs)-Fähigkeiten für eine Vielzahl von Problemen geweckt. Eine besonders bekannte Sprache, die skalierbare Reasoning-Techniken unterstützt, ist Vadalog. Sie unterstützt fortgeschrittene Reasoning-Fähigkeiten wie existentielle Quantifizierung, Rekursion, Aggregation, probabilistisches Reasoning sowie verschiedene Datenquellen und stellt ihren Wert in zahlreichen Finanzanwendungen wie Unternehmenskontrolle (Company Control) und Golden Power Checks unter Beweis.

In all diesen Anwendungen ist Zeit eine entscheidende Dimension, um ein tieferes Verständnis struktureller Veränderungen zu erlangen. Bisher fehlt Vadalog jedoch die Unterstützung mit temporaler Information umzugehen, wodurch es nur eingeschränkt im zeitlichen Kontext anwendbar ist. Das Fehlen einer solchen Funktionalität in Vadalog wird durch DatalogMTL unterstrichen, einer vor kurzen erschienen Erweiterung von Datalog mit Operatoren aus der metrischen temporalen Logik, welche das zeitliche Reasoning im Zusammenhang mit Stream-Reasoning wieder populär gemacht hat. Da DatalogMTL jedoch lediglich eine Erweiterung von Datalog ist, fehlen der Sprache viele der Fähigkeiten, die für Knowledge Graph Reasoning erforderlich sind. Infolgedessen wird in dieser Dissertation DatalogMTL mit einem Fokus auf deren Anwendung auf Knowledge Graphen untersucht.

Zuerst werden Erweiterungen von DatalogMTL, nämlich Aggregation und existentielle Quantifizierung, untersucht, die für zahlreiche Data-Science-Workflows von grundlegender Bedeutung sind. Im Detail wird die formale Syntax und Semantik definiert, verschiedene Möglichkeiten der Aggregation entlang der Zeitachse erkundet sowie eine „natürliche“ als auch eine „einheitliche“ Semantik der existentiellen Quantifizierung untersucht.

Im Anschluss wird ein neuartiger Benchmark-Generator vorgestellt, der als erster seiner Art die Erstellung von Benchmarks mit metrischer temporaler Logik im Zusammenspiel mit rekursiven Abfragen, Aggregation und existentieller Quantifizierung ermöglicht. Auf diese Weise wird gezielt das Generieren von Instanzen zum Testen bestimmter Szenarien und Grenzfälle ermöglicht.

Danach wird Vadalog mit einer vollständig entwickelten Architektur erweitert, die die Fähigkeit besitzt mit metrischer temporaler Logik Reasoning zu betreiben. Das System

wird mit realen Instanzen sowie mit Benchmarks, die von dem in dieser Arbeit vorgestellten Generator generiert wurden, evaluiert. Die Ergebnisse zeigen, dass das vorgestellte System aktuelle Lösungen in den meisten Szenarien übertrifft.

Abschließend wird die Verwendung von DatalogMTL als Spezifikationssprache von Smart Contracts definiert. Dadurch wird die Verwendung von DatalogMTL für dezentralisierte Finanzen (DeFi) ermöglicht, was eine interessante Domäne für Knowledge Graph Reasoning darstellt.

Abstract

The rise of knowledge graphs has sparked great interest in providing scalable and efficient reasoning capabilities for a variety of problems. A particularly prominent language supporting scalable reasoning techniques is Vadalog, which supports advanced reasoning capabilities such as existential quantification, recursion as well as aggregation, probabilistic reasoning, and various data sources and has demonstrated its value in numerous financial applications, including company control and golden power checks.

In all of these applications, time is a critical dimension to gain a deeper understanding of the structural changes. However, so far, Vadalog is missing the support for dealing with temporal information, limiting its applicability in temporal contexts. The absence of such functionality is emphasized by the resurgence of temporal reasoning in the context of stream reasoning through DatalogMTL, an extension of Datalog with operators from the metric temporal logic. Yet, since DatalogMTL is a merely extension of Datalog, it lacks many of the capabilities necessary for knowledge graph reasoning. As a result, in this thesis, we conduct the first study on how to extend DatalogMTL towards its application in knowledge graph reasoning.

For this purpose, we first study extensions of DatalogMTL, namely aggregation and existential quantification, which are fundamental to numerous data science workflows. In detail, we define formal syntax and semantics, explore different possibilities for aggregating along the timeline as well as examine a natural as well as a uniform semantic for existential quantification.

Subsequently, we present a novel benchmark generator that is the first of its kind which is capable of supporting the generation of benchmarks for metric temporal logic, together with recursive queries, aggregation and existential quantification. This allows us to generate targeting instances for testing specific scenarios and edge cases.

Afterwards, we augment Vadalog with the ability to reason with metric temporal logic providing a fully engineered reasoning architecture. We evaluate the system with benchmarks generated from our generator as well as from real-world instances. The results show that our system outperforms state-of-the art solutions in most of the scenarios.

Finally, we discuss the usage of DatalogMTL as specification language of smart contracts, enabling the use of DatalogMTL for decentralized finance, an interesting domain for knowledge graph reasoning.

Contents

CHAPTER 1

Introduction

The rise of *knowledge graphs* has sparked great interest in providing scalable and efficient reasoning capabilities for a variety of problems. In detail, knowledge graphs represent entities and relations between them in a graph structure and enrich them with semantic information. This enables a variety of applications such as in the financial domain [BMNS20] (e.g., company takeover prediction or anti-money laundering mechanisms), recommender systems [WHC+19] (e.g., for movies or songs), in the medical domain [EMSW14, RHT+17] (e.g., to map diseases to symptoms), or in cyber security [KEKE19] (e.g., for attack prediction).

An essential ingredient in all of these applications is the use of *reasoning* technologies [GO22]. Deductive reasoning enables us to use the specified semantic information inside the knowledge graph to extract logical consequences with the help of a reasoning engine, such as Vadalog [BSG18]. Vadalog is a prominent Datalog-based reasoner that combines high expressive power with impressive scalability. These are fundamental capabilities for handling a vast amount of data that are present, e.g., with the success of Internet of Things, in nearly all domains nowadays.

Recent advances in knowledge graphs emphasize the inclusion of *time*-related properties as additional metaknowledge to meet the requirements of a rapidly changing world, where events are valid at multiple time points as well as only for a specific interval. Indeed, the studying of temporal information is not a new topic and has been considered in the context of temporal logics for a long time [Koy90, GFAA03] (for example, with the well-known linear temporal logic or computation tree logic). In addition to these studies, such languages have also been regularly considered as an extension to Datalog [Cho90, GGV02, BKK+17a], providing first results for an integration in knowledge graph systems.

Yet, while there exists a long history of temporal logics, so far, to the best of our knowledge, the deductive temporal reasoning capabilities are limited for knowledge graphs: In particular, there is a missing support of an efficient and highly expressive

temporal logic in current knowledge graph-optimized reasoning systems that integrates the desired properties of inventing new values (i.e., *existential quantified variables*) for creating new edges and nodes in the knowledge graph, as well as *arithmetic and aggregation* for data analysis. At the same time, the inductive approaches, in particular embedding based deep-learning approaches, are promoting their research on temporal reasoning [GO22], creating an imbalance of the available tools in temporal reasoning between inductive and deductive reasoning frameworks. This creates an urgent need for the deductive community to gain momentum, make progress and sharpen their available tools to have a good mixture between the inductive and deductive approaches in place.

Taking this imbalance into account, in this thesis, we investigate the temporal dimension of logical reasoning in knowledge graphs and reduce the gap of available tools between deductive and inductive approaches. Towards this, in this introductory chapter, we introduce knowledge graphs in a broader context before introducing the research questions of the thesis targeting exactly the gap, namely the functionality of existing logic-based languages and the development and evaluation of a reasoning engine with native support of temporal information. In more detail, we are going to consider the following.

Motivation and Context. In this part, we establish the context of this thesis, progressively introducing all concepts important to it:

- Knowledge Graphs (Section 1.1.1)
- Logic-based Reasoning in KGs (Section 1.1.2)
- Temporal Reasoning in KGs (Section 1.1.3)

Challenges and Contributions. In this part, we establish the main challenges and contributions of the thesis. This includes formulating our research questions, and is divided into four areas:

- **Functionality** – Providing needed functionality to languages (Section 1.2.1)
- **Evaluation** – Furthering principled development and evaluation (Section 1.2.2)
- **System** – Creating a temporal-aware reasoning engine (Section 1.2.3)
- **Applications** – Bringing together these points into applications (Section 1.2.4)

Structure and Publications. In this part, we present an outline of the thesis as well as an overview of the published papers (Section 1.3)

1.1 Motivation and Context

In this section, we introduce to a reader unfamiliar with the domain, the main concepts of knowledge graphs. We start with a generic definition and provide an overview of different artificial intelligence tools for knowledge graphs in Section 1.1.1. Then, in Section 1.1.2 we focus on logical reasoning, a field of artificial intelligence. In particular, we focus on the language Datalog which we use throughout the thesis. We conclude this primer in Section 1.1.3 with a brief introduction to temporal reasoning with Datalog.

1.1.1 Introduction to Knowledge Graphs

In 2012, Google [Sin12] introduced the term knowledge graph in the context of making search more effective. They describe knowledge graphs as "a graph that understands real-world entities and their relationships to one another: things, not strings". Since then, other researchers have proposed various definitions for knowledge graphs [Ber19] and still new definitions are proposed regularly since "the technology used to create them is rapidly changing" [Dee]. For instance, Ehrlinger and Wöß [EW16] define knowledge graphs as "A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge" and Bellomarini et al. [BFGS19] as a characterization of three components: (i) a ground extensional component representing the data in a graph-like structure, (ii) an intensional component as a set of inference rules over the constructs of the ground extensional component and (iii) a derived extensional component as a result of the "reasoning" process (i.e., the application of the inference rules over the ground extensional component).

In other words, one typically considers arbitrary (semi-structured) data-sources, such as relational databases, NoSQL stores, RDF stores or graph databases, which are mapped into a graph-like structure on which together with given domain knowledge certain "functions" are applied to either derive new knowledge (in the sense of extracting unknown domain knowledge, e.g., rule learning) or new data (by the application of reasoning techniques). Note that the mapping does not imply that the data is stored in a graph structure and the graph is seen here only as a view on the data in a unified data model (cf., Chen [Che76]). In the remainder of this section, we will briefly introduce such different functions.

Example 1.1. Given a relational database in Table 1.1 that contains a table that stores the stake of shares of companies owned by an entity (i.e., a person or a company), where numbers indicate people and letters companies. The corresponding graph view is given in Figure 1.1a, where orange nodes represent people, blue nodes companies and the stake of shares are provided as properties of the edges connecting entities.

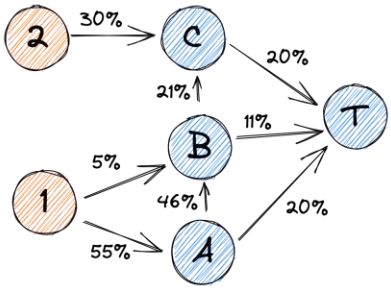
Knowledge Graph Embeddings

Knowledge graph embeddings are numerical representations, e.g., a d-dimensional vector of real numbers, of the entities (nodes) and relations (edges) of a knowledge graph in

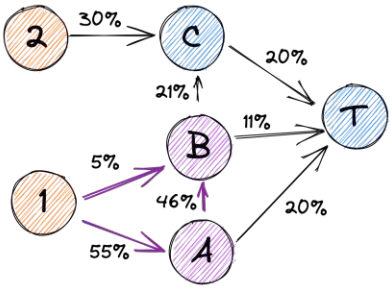
Table 1.1: Percentage of shares of a company per entity

Source	Target	Shares
1	A	55%
1	B	5%
2	C	30%
A	T	20%

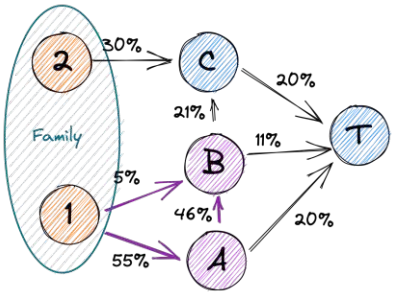
Source	Target	Shares
A	B	46%
B	T	11%
B	C	21%
C	T	20%



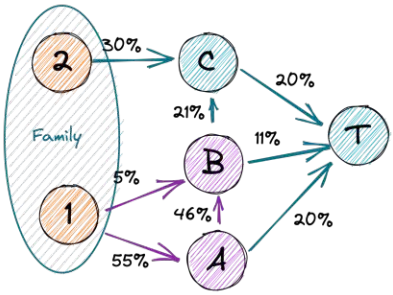
(a) Graph representation of Table 1.1



(b) Inference of company control per person



(c) Detection of family clusters



(d) Inference of company control per family

Figure 1.1: Example of (family) company control based on Table 1.1.

a low-dimensional space, e.g., where the dimensionality d is many orders of magnitude smaller than the number of nodes and edges. They are used for tasks such as link or entity prediction (given a triple $(h, r, ?)$, $(h, ?, t)$, or $(?, r, t)$, where h is a head entity, t a tail entity, and r a relation between the head and the tail, predict the missing component "?") as well as triple classification (given a triple, is it valid) [CLMS21, WQW21].

Example 1.2. Consider the graph illustrated in Figure 1.1c, where the persons 1 and 2 are linked as family members. A family is a group of closely related people with shared interests, which is critical to detect in settings such as anti-money laundering or fraud detection [ABI+20]. One approach to detect such family links is by the usage of knowledge graph embeddings, which are capable of learning that nodes 1 and 2 belong to the same family. For example, by triple classification it can determine whether the given triple $(1, f, 2)$ is valid, where f represents the family relation.

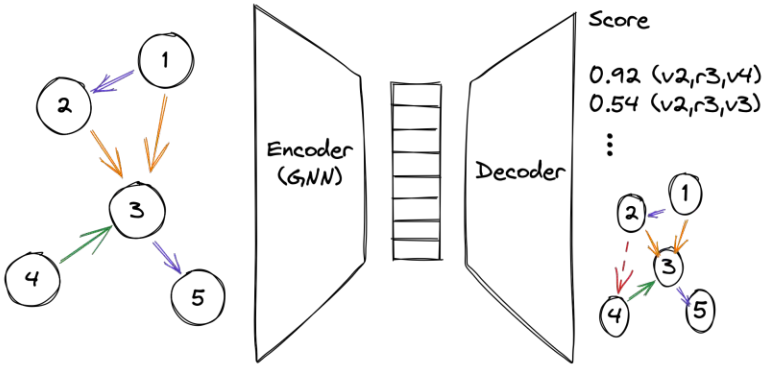


Figure 1.2: Graph neural networks used as an encoding layer for link prediction.

Knowledge graph embeddings are formulated as an optimization problem. The aim is to minimize the loss in such a way that the scoring function of the model provides a high score to positive triples (i.e., triples that are very likely to be true). Typical models vary in their representation space, their scoring functions which are used to determine the plausibility of a triple, their exact encoding model, as well as their possible auxiliary information that is incorporated into the embedding method [WQW21, JPC+22].

In the literature, models are often classified by their encoding model. One usually distinguishes between linear models, factorization, and neural networks [CLMS21, WQW21]. Linear models, such as TransE [BUG+13] or RotatE [SDNT19] usually express the scoring function by representing the plausibility of a fact by the translational distance between entity and relation embeddings. Models based on matrix factorization, such as DistMult [YYH+15] or Complex [TWR+16] usually follows the semantic matching pattern and thus use a similarity-based scoring function. Neural networks, such as ConvE [DMSR18] or HypER [BAH19] are able to propagate neighborhood information and use the recent advancements in neural networks [WQW21].

Graph Neural Networks

Graph neural networks (GNNs) are related to knowledge graph embeddings but serve a different purpose. While knowledge graph embeddings are transductive (i.e., ineffective for unseen data), GNNs are inductive (i.e., are able to reason about unseen entities). In order to accomplish this, GNNs usually learn high-level representations of nodes by considering the properties of entities and their interaction with their neighborhood in the graph. That is why GNNs are seen as ideal candidates for an encoding layer (i.e., converting input to a low-dimensional representation) for nodes before applying a decoding layer (i.e., converting a low-dimensional representation to an output) to accomplish a specific task (e.g., an embedding task such as link prediction) as visualized in Figure 1.2. Other tasks include graph and node classification as well as graph generation [WPC+21, YKS+22].

GNNs are again classified based on their method. Recurrent GNNs, such as Graph-

ESN [GM10], propagate neighborhood information until a stable fix point is obtained. Convolutional GNNs redefine the concept of convolution neural networks (CNNs) from the computer vision domain. They are either based on the spectral graph theory, such as GCN [KW17], where filters are applied to remove noises from graph signals or spatially based, such as GraphSage [HYL17], which inherits the concept of information propagation from recurrent GNNs. Graph autoencoders, such as GraphRNN [YYR+18] are utilized for unsupervised learning, where no ground truth for data exists [WPC+21].

Logic-based Reasoning

The term logic-based reasoning subsumes the different logic-based knowledge representation and reasoning languages, such as Cypher [FGG+18], Datalog [MTKW18], ASP (answer set programming) [EIK09], SQL [CB74], SPARQL [PAG09] and description logic [BCM+03]. While certain languages, such as SQL, SPARQL and Cypher, have moved away from a rule-like syntax well-known from first order logic [Smu95], the most relevant distinction between the languages is their supported feature set and the resulting computational complexity of the language. For example, languages differ by the support of disjunction in the rule head (e.g., $\mathit{control}(\text{Src}, \text{Tgt})$ in Example 1.3 is considered as head of the rule), the supported level of recursion, the supported type of negated values, or the support of value invention (i.e., existential quantification or in short existentials).

Yet, the reasoning with first order logic (or a variant thereof) does not yield the required expressive power to solve certain problems. Therefore, extensions such as the family of modal logics [Gar21] (e.g., deontic logic for legal reasoning or temporal logic for time-based reasoning) as well as probabilistic logic [Nil86] (i.e., to reason over uncertain data) have been established. Temporal logic, the relevant extension for this thesis, will be introduced in Section 1.1.3.

Example 1.3. We continue our running example from the financial domain. Direct company control [BSG18] is defined as the direct ownership of more than 50% of a company by a single entity. As an example, consider Figure 1.1b, in which person 1 personally owns company A with 55% of the shares. The rule of direct company control can be encoded as follows in some selected logic-based languages:

Language	Query
SQL	SELECT Src, Tgt INTO control FROM own WHERE S $>$ 0.5
Cypher	MATCH (Src:Company)-[o:Own]->(Tgt:Company) WHERE o.S $>$ 0.5 CREATE (Src)-[:Control]->(Tgt)
Datalog/ASP	$\mathit{control}(\text{Src}, \text{Tgt}) \leftarrow \mathit{own}(\text{Src}, \text{Tgt}, \text{S}), \text{S} > 0.5$

Neuro-Symbolic AI

The objective of neuro-symbolic AI is to bring together neuro (also called subsymbolic) AI (i.e., machine-learning based methodologies such as knowledge graph embeddings or graph neural networks) and symbolic AI (i.e., logic-based reasoning). Kautz [Kau22] proposed a taxonomy that consists of six different types for the combination of the two fields, which are based on whether the input or output is symbolic or subsymbolic, and how the symbolic or subsymbolic procedure invokes/includes the subsymbolic/symbolic portion of the program.

In the context of knowledge graph reasoning, two techniques gained significant attention. Knowledge injection, such as SimplE [KP18] and BoxE [ACLS20], aims to inject logical rules during the training of a knowledge graph embedding such that the model conforms to these rules. Depending on the model, different kind of rules are supported (e.g., hierarchical, symmetric, inversion and so on). Rule mining approaches, such as AMIE [GTHS15] and AnyBURL [MCRS19], aim to learn/extract rules from a given knowledge graph.

1.1.2 Logical Reasoning with Datalog in Knowledge Graphs

In the previous section, we provided a general overview of the current reasoning techniques in knowledge graphs. As this thesis contributes to logical reasoning, particularly by contributing to Datalog extensions, we introduce Datalog in this section in further detail.

Datalog is a fully declarative programming language. It is frequently used as a query language in deductive databases, but also for data integration, information extraction and of course as a knowledge representation language in knowledge graphs [MTKW18]. Particularly for large graph structures, Datalog offers, with the support of recursion and stratified negation (a limited type of negation that prohibits the use of negation in recursion), the essential capabilities for efficient and scalable reasoning over such graph structures while still having PTIME data complexity. With its extensions for existential quantification, subsumed under the term Datalog$^{\pm}$, the research community seeks to extend the functionality of Datalog (the + term), while restricting the language (the - term). With this, even more essential capabilities of knowledge graph reasoning are supported, namely the creation of new nodes and edges[1], while still maintaining PTIME data complexity [GP15, BSG18].

In general, Datalog consists of a set of rules (a program) and a database of facts. In plain Datalog (i.e., Datalog without existential quantification or negation) rules are function-free Horn clauses (i.e., a disjunction of literals with at most one positive literal in the clause where a literal is either a variable or a constant from a domain). These Horn clauses are usually written as implications [MTKW18] where we sometimes replace the conjunctions with a comma.

[1] Note that the solely extension by existential quantification would yield undecidability.

Example 1.4. Assume a database with a single fact $father(\text{john}, \text{anna})$ and a program with a single rule that models that if X is a father of Y, then Y is the child of X. This may be stated as a Horn Clause in disjunctive form

$$\neg father(\text{X}, \text{Y}) \vee child(\text{Y}, \text{X})$$

or in implication form, which is usually used when writing rules

$$child(\text{Y}, \text{X}) \leftarrow father(\text{X}, \text{Y})$$

The reasoning process then derives the fact $child(\text{anna}, \text{john})$.

The objective of the reasoning process is to find a model (i.e., a new database of facts) that satisfies the program with a given database. In Datalog, there always exists a unique minimal model that extends the given database. This unique model can be defined in Datalog with three distinct, but equivalent semantics. From an applied perspective the operational semantics, one of the three aforementioned equivalent semantics, offers the most benefits since it provides an algorithm for computing the unique minimal model. The idea of the operational semantics is to begin with the given database, and repeatedly add the heads of an instantiated rule (i.e., where variables are replaced with constants from the domain) whose body is satisfied, until no more rule head can be added (i.e., a fixpoint is reached) [AHV95].

Example 1.5. Consider Figure 1.1b once more. This time we are interested not just in direct company control, but also in indirect controls over companies. This necessitates an additional rule of the form [BSG18]

$$control(\text{X}, \text{Z}) \leftarrow control(\text{X}, \text{Y}), own(\text{Y}, \text{Z}, \text{V}), \text{S} = \text{sum}(\text{V}), \text{S} > 0.5$$

The use of this new rule derives the control of B by 1 only after deriving that A is controlled by 1 (by direct company control). Assuming that a family controls 100% of person 1 and 2 in Figure 1.1d, we see that several iterations to reach a fixed point are required to determine that the family controls every node in the graph. Note that we already used advanced capabilities of Vadalog (introduced next) by utilizing its aggregation capability for computing company control.

One of the most famous Datalog-based languages for reasoning in knowledge graphs is Vadalog [BSG18], which extends Warded Datalog$^{\pm}$ [GP15] with additional capabilities such as aggregation and arithmetic. Warded Datalog$^{\pm}$ is conceptualized in such a way that it limits the use of existential quantification while fully supporting Datalog with stratified negation.

1.1.3 Temporal Reasoning with Datalog

So far, we focused on existing capabilities of Vadalog and have neglected to include the temporal dimension in our discussion which we will now do so.[2] Temporal logic, part of the modal logic family, offers one possibility to reason over time via modalities such as "always", "until" or "later". It was introduced as a tool for formal verification and its most well-known logics are computation tree logic (CTL), and linear temporal logic (LTL) [GFAA03]. In detail, LTL extends first order logic with temporal operators that state whether some formula holds forever in the future ($\boxplus$)/past ($\boxminus$), sometimes in the future ($\diamondplus$)/past ($\diamondminus$), at the next ($\oplus$)/previous ($\ominus$) time point or that something has to hold at least until ($\mathcal{U}$)/since ($\mathcal{S}$) some other formula is true. For further details, we refer a reader unfamiliar with LTL to the summary [GR22] to find further information.

In the thesis, we focus on metric temporal logic (MTL) [Koy90] over continuous semantics, since it has recently demonstrated its successful application in a variety of complex real-time and real-world use cases together with Datalog [WKCG19, BKK$^+$17a]. MTL is an extension of LTL in which the temporal modalities are substituted with time-constrained counterparts, which is heavily required in real-time scenarios. This means that instead of stating that something will hold in the future, or something holds the next state, as in LTL, one states that something will hold for the next x to y time units (see Example 1.6).

Example 1.6. Consider the LTL formula $\oplus\oplus\oplus A \vee \oplus\oplus\oplus\oplus A$ saying that A will hold either three or four states in the future. This is equivalent to the MTL formula $\diamondplus_{[3,4]} A$ interpreted over the integer timeline, where $\diamondplus$ states that A holds at some point in the future and the interval $[3, 4]$ restricts the range to exactly when this will be the case (i.e., either three or four states in the future).

One popular branch that supports query answering with MTL is DatalogMTL [BKR$^+$18], an extension of plain Datalog with MTL operators. Although the data complexity of this language is PSPACE-complete, different restrictions such as non-recursive programs [BKK$^+$17a] or the restriction of temporal operators [WCGKK20b] have been studied to achieve tractability, thus making this language attractive for knowledge graph reasoning.

In more detail, DatalogMTL[3] extends each fact with a time point where the fact is valid and rules with the operators diamond plus ($\diamondplus$)/minus ($\diamondminus$) to express that something holds at some time point in a given interval in the future/past, box plus ($\boxplus$)/minus ($\boxminus$) to express that something holds at all time points in a given interval in the future/past, and until ($\mathcal{U}$)/since ($\mathcal{S}$) to express that something holds in a given interval until/since some time point in the future/past.

[2] Due to the emphasis of this thesis on logic-based temporal reasoning, we solely introduce the temporal domain for logical reasoning. For completeness, we would like to mention that there is active research in the areas of temporal knowledge graph embeddings and temporal graph neural networks. For further information, we direct the reader to one of the several surveys [WPC$^+$21, JPC$^+$22] in this field.

[3] Note that we formally introduce the syntax and semantics in Chapter 2.

Example 1.7. A trader is flagged as suspicious for five minutes, if the trader produces new trade offers with a maximum gap of five seconds throughout the course of the previous minute.

$$\boxplus_{[0,300]}\, suspicious(\mathrm{t}) \leftarrow \boxminus_{[0,60]}\, \diamondminus_{[0,5]}\, tradeOffer(\mathrm{t}, \mathrm{o})$$

With this brief introduction to temporal reasoning, we conclude our primer to knowledge graph reasoning.

1.2 Main Challenges and Contributions

This thesis aims to enable reasoning with DatalogMTL in knowledge graph systems, or in other words to extend DatalogMTL with reasoning capabilities provided in Vadalog for the non-temporal context. Towards this, we have to resolve a number of open challenges, where some of them are addressed in this thesis. That is, in order to support temporal reasoning in knowledge graphs, we have to bring together different capabilities to create a *principled, scalable system*:

- *Recursive reasoning.* Support of full recursion, e.g., to enable the traversal of cyclic graph structures (as provided by Datalog).
- *Ontological reasoning.* Support of existential quantification to create new knowledge in a knowledge graph (as provided by Datalog$^{\pm}$; not considered for DatalogMTL).
- *Temporal reasoning.* Support for temporal logic and reasoning to enable efficient reasoning over temporal relationships in a knowledge graph (as provided by DatalogMTL).
- *Aggregate reasoning.* Support for aggregation that is compatible with both, the temporal, as well as the recursive aspects (as discussed for Datalog$^{\pm}$, but no such results are known for DatalogMTL).

Specifically, the contributions of this thesis fall into four categories, all of which have the common theme of facilitating temporal reasoning in knowledge graphs. The categories are visualized in Figure 1.3: (i) the enhancement of DatalogMTL with additional functionality, (ii) the provision of benchmark data for evaluating systems, (iii) system design, and (iv) the development of novel real-world applications on top of the three established pillars.

In the following, we discuss the main contributions of each category, with further information and results then provided in the respective chapters.

1.2.1 Functionality

As DatalogMTL is built upon plain Datalog, the existing features defined for Datalog have to be transformed and extended to the temporal context. For reasoning in knowledge

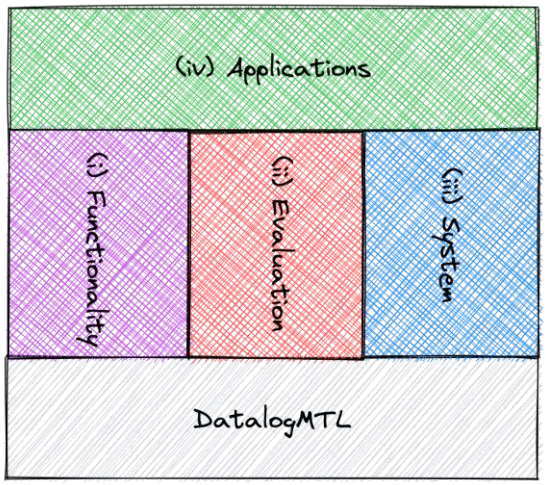


Figure 1.3: Overview of contributions

graphs, we have identified two key elements that have to be extended for temporal reasoning: monotonic aggregation and reasoning with existential quantification.

Monotonic Aggregation. Extending the support of monotonic aggregation as used in Vadalog to the temporal domain is the first missing element for many data science applications (e.g., the computation of temporal company control). This requires support for several types of temporal aggregation, such as moving window or span temporal aggregation, well-known from time-series databases [GBJ18] together with the full support over cyclic structures. The aim of this extension is to utilize existing techniques from Datalog, which raises the first research question:

> **Research Question 1.** In which form can monotonic aggregation, as used in Datalog, be used over temporal data, i.e., in DatalogMTL?

In this thesis, we provide an answer to this question and demonstrate that we are able to reduce the different types of aggregation to a time-point aggregation, which can be interpreted as a monotonic aggregation in Datalog. For this, we introduce an additional granularity operator for the span temporal aggregation, which extends time points to the complete unit of interest (e.g., the operator applied in "month-mode" on the 3rd of April yields the range from the 1st of April to the 30th of April). Yet, aggregating every time-point is inefficient since there are typically constant data periods, where the same aggregation would have to be executed for each time point. Therefore, we present efficient methods that utilize the interval information by grouping constant periods and evaluating them together. In addition, we also identify the importance of analyzing behavior across time to detect trends. We focus on data that is monotonically increasing or decreasing and generalize our definitions for supporting such operations.

Existential Quantification. The second missing element is the invention of new values. It can be seen that the extension with existential quantification to the temporal domain creates a number of interpretations for an existential quantification (e.g., per time point, per concrete interval, and so on). The two edge cases we consider for the existential value are (i) per time point separately (called the natural semantics), and (ii) the same for all time points (called the uniform semantics). Especially, the latter one appears to be extremely useful, as throughout an activity (e.g., a flight) something (e.g., the airplane) usually does not change. Moreover, we know from Datalog, that the complexity of existential quantification varies depending on the chosen restriction, which poses the following research question:

> **Research Question 2.** What is the complexity for different restrictions of existential quantification? Is there a way to adapt the reasoning process towards the usage of existential quantification?

In this thesis, we present complexity results for different examined existential quantification fragments. While the majority of the fragments have been identified as undecidable, we identify an interesting decidable fragment for the uniform semantics for weakly-acyclic programs, for which we suggest an adapted reasoning method.

1.2.2 Evaluation

The combination of recursion, temporal reasoning using MTL, and aggregation is particularly relevant for temporal knowledge graphs, yet it has not been the focus of existing benchmark datasets. To test and evaluate systems of these kinds, we have to develop our own benchmarks, which leads us to the following research question:

> **Research Question 3.** What queries should a benchmark for DatalogMTL include? What datasets should be used for the benchmark?

In this thesis, we propose a new highly flexible and user-adaptable benchmark generator that is aware of the temporal domain and is able to generate benchmarks that take into account temporal operators, monotonic aggregation and recursive structures. We also add support for existential quantification to support the evaluation of weakly-acyclic uniform DatalogMTL programs.

1.2.3 System

The last step towards a knowledge graph reasoning system that facilitates efficient reasoning across the temporal domain is the construction of the system itself, which poses the following research question:

> **Research Question 4.** How is the performance of DatalogMTL in comparison with different state-of-the-art systems and between different implementation choices?

In this thesis, we propose Temporal Vadalog, a fully engineered system that supports reasoning with DatalogMTL including temporal aggregation and existential quantification. Our system adapts the pipes-and-filters architecture used in Vadalog and optimizes it for temporal reasoning. We evaluate our system with different competitors in its domains and show that our system outperforms existing DatalogMTL reasoners.

1.2.4 Applications

DatalogMTL provides several opportunities for real-world applications. We were particularly interested in DatalogMTL as a smart contract language since with the encoding of smart contracts through DatalogMTL we are able to provide explainable insights into the behavior of smart contracts[4]. This opens the possibility to better understand and automatically deduce connections in the transaction flow, which is typically modelled in a graph-like structure. In this regard, we investigate the following research question:

> **Research Question 5.** Is it possible to encode a variety of typical smart contract use cases in DatalogMTL? How can we enable DatalogMTL-based smart contracts on major blockchain platforms?

In this thesis, we elaborate on the necessary principles for smart contracts written in DatalogMTL. Particularly, we concentrate on the design of a language that is able to express typical smart contract patterns, such as state machines and ERC20-tokens, in DatalogMTL and can be converted to a Solidity smart contract, the most popular language for developing smart contracts in Ethereum, for which we also suggest a preliminary version of a translation engine.

1.3 Structure of this Thesis

The remainder of this thesis is organized as follows:

- **Chapter 2** introduces the preliminaries and key notions relevant for the remainder of this thesis, including Datalog, DatalogMTL and monotonic aggregation. We further discuss the current state of the art in temporal reasoning, with an emphasis on related work related to the topic of the thesis.
- **Chapter 3** addresses the first main part of the thesis by introducing important additional functionalities to DatalogMTL required for the use cases in knowledge graphs. We begin with monotonic aggregation and analyze how DatalogMTL can be extended with aggregation. We consider the three well-known types of aggregation: per time point, per moving window and spanning time aggregation. We define

[4] We explain the details of blockchains and smart contracts in the corresponding section. For now, it suffices to know that smart contracts are executable code that facilitate the process of executing and enforcing the terms of an agreement between (untrusted) parties and that each execution of a smart contract is stored as a transaction on the blockchain.

syntax and semantics for temporal monotonic aggregation and provide methods for the four commonly discussed aggregate types, namely, *min*, *max*, *count* and *sum*. In addition, we analyze an additional form of aggregation, namely along the time-axis for which we consider monotonic increasing or decreasing trends. Then, we discuss existential quantification in DatalogMTL. We present formal definitions of the natural and uniform semantics and provide complexity results of the studied fragments as well as an reasoning algorithm for weakly-acyclic programs under uniform semantics.

- **Chapter 4** introduces a benchmark generator for DatalogMTL queries that allows us to evaluate our system. We first observe the key requirements of such a generator and then suggest a generation approach to construct arbitrary DatalogMTL programs. On top of that, we introduce a graphical user interface that allows users to build benchmarks without understanding the details of the generator.

- **Chapter 5** introduces the Temporal Vadalog System. We describe a fully engineered time-aware pipes-and-filters architecture that enables temporal reasoning with MTL operators, including stratified negation, various merge strategies for merging overlapping intervals, termination strategies as well as aggregation functions and arithmetic. We also discuss an interface between temporal and non-temporal reasoning. Finally, we provide an evaluation by comparing the implementation to state-of-the-art systems.

- **Chapter 6** introduces the modelling of smart contracts with DatalogMTL. We propose a formal definition of a DatalogMTL smart contract, describe the interaction with existing blockchains, provide a case study, and introduce a translation engine to Solidity.

- **Chapter 7** concludes this thesis with a summary of the results and a discussion of open problems and future research directions.

The thesis is based on the following peer-reviewed papers during my doctoral studies, which have been accomplished in joint work with different collaborators:

- Luigi Bellomarini, Markus Nissl, and Emanuel Sallinger. Monotonic Aggregation for Temporal Datalog. *RuleML+RR 2021, Rule Challenge Paper.*
  Winner of the Rule Challenge Award

- Luigi Bellomarini, Markus Nissl, Emanuel Sallinger. iTemporal: An Extensible Generator of Temporal Benchmarks. *ICDE 2022.*

- Luigi Bellomarini, Livia Blasi, Markus Nissl, and Emanuel Sallinger. The Temporal Vadalog System. *RuleML+RR 2022.*
  Invited for Theory and Practice of Logic Programming

- Markus Nissl and Emanuel Sallinger. Modelling Smart Contracts with DatalogMTL. *EcoFinKG@EDBT/ICDT 2022.*
- Markus Nissl and Emanuel Sallinger. Towards Bridging Traditional and Smart Contracts with Datalog-based Languages. *Datalog 2.0 2022.*
- Matthias Lanzinger, Markus Nissl, Emanuel Sallinger, and Przemysław Wałęga. Temporal Datalog with Existential Quantification. *IJCAI 2023, to appear.*

In addition, I have completed several other peer reviewed publications during my doctoral studies which are beyond the scope of this thesis:

- Luigi Bellomarini, Marco Benedetti, Stefano Ceri, Andrea Gentili, Rosario Laurendi, Davide Magnanimi, Markus Nissl, and Emanuel Sallinger. Reasoning on Company Takeovers during the COVID-19 Crisis with Knowledge Graphs. *RuleML+RR 2020, Industrial Paper.*
- Luigi Bellomarini, Davide Magnanimi, Markus Nissl, and Emanuel Sallinger. Neither in the Programs Nor in the Data: Mining the Hidden Financial Knowledge with Knowledge Graphs and Reasoning. *MIDAS@PKDD/ECML 2020.*
- Luigi Bellomarini, Markus Nissl and Emanuel Sallinger. Blockchains as Knowledge Graphs - Blockchains for Knowledge Graphs (Vision Paper). *KR4L@ECAI 2020.*
- Markus Nissl, Emanuel Sallinger, Stefan Schulte, and Michael Borkowski. Towards Cross-Blockchain Smart Contracts. *DAPPS 2021.*
- Nicolas Ferranti, Astrid Krickl, and Markus Nissl. Knowledge Graphs: Detection of Outdated News. *ISWC 2021, Poster Paper.*
- Luigi Bellomarini, Giuseppe Galano, Markus Nissl, and Emanuel Sallinger. Rule-based Blockchain Knowledge Graphs: Declarative AI for Solving Industrial Blockchain Challenges. *RuleML+RR 2021, Industrial Paper.*
- Luigi Bellomarini, Lorenzo Bencivelli, Claudia Biancotti, Livia Blasi, Francesco Paolo Conteduca, Andrea Gentili, Rosario Laurendi, Davide Magnanimi, Michele Savini Zangrandi, Flavia Tonelli, Stefano Ceri, Davide Benedetto, Markus Nissl, and Emanuel Sallinger. Reasoning on company takeovers: From tactic to strategy. *Data & Knowledge Engineering, Vol. 141 2022.*
- Teodoro Baldazzi, Luigi Bellomarini, Markus Gerschberger, Aditya Jami, Davide Magnanimi, Markus Nissl, Aleksandar Pavlovic, and Emanuel Sallinger. Vadalog: Overview, Extensions and Business Applications. *Reasoning Web 2022.*
- Zilu Tian, Peter Lindner, Markus Nissl, Christoph Koch, Val Tannen. Generalizing Bulk-Synchronous Parallel Processing for Data Science: from data to threads and agent-based simulations. *Proceedings of the ACM on Management of Data, Vol. 1, Issue 2 2023*

CHAPTER 2

Preliminaries and State of the Art

In this chapter we present the relevant definitions from the literature used in this thesis. We begin by introducing Datalog in Section 2.1 and extend it with the notions of stratified negation and existential quantification in the subsections. In Section 2.2 we introduce the notions for DatalogMTL by building on the introduced notions in Section 2.1. Then, we present an overview of the various types of temporal aggregation in Section 2.3. In Section 2.4 we introduce the main concepts of the reasoning framework Vadalog. Finally, we discuss the current state of the art in temporal reasoning in Section 2.5.

2.1 Datalog

In this section we present Datalog. We first discuss the syntax and semantics of Datalog. Then, we extend Datalog with stratified negation as well as existential variables. We primarily follow the definitions of the book by Abiteboul et al. [AHV95], enriched with some additional information [GOPS12, GHLZ13, GP15].

Syntax Assume a function-free first-order signature, i.e., disjoint sets of logical symbols containing only relation and constant symbols (i.e., no function symbols) and a mapping that assigns an arity to each relation symbol, over a domain Dom consisting of countably infinitely constants. A *term* is a constant from Dom (represented by alphanumeric strings which are either numeric or begin with a lower-case letter in the following) or a variable (represented by alphanumeric strings beginning with an uppercase letter in the following). An *atom* is an expression of the form $P(\mathbf{s})$, where P is a predicate of arity $n \geq 0$ and $\mathbf{s}$ is a n-tuple of terms. A *literal* is an atom. A *rule* r is an expression of the form

$$H \leftarrow B_1 \wedge \cdots \wedge B_k, \qquad \text{for } k \geq 0, \tag{2.1}$$

where $B_1 \wedge \cdots \wedge B_k$ and H are literals. The conjunction $B_1 \wedge \cdots \wedge B_k$ of the rule is called the body, denoted $B(r)$, and H is called the head, denoted $H(r)$. A rule is *safe* if all its

variables occur in the body. A *program* Π is a finite set of safe rules. An expression is *ground* (e.g., atom, rule, program) if it contains no variables. A *fact* is a ground atom. A *dataset* $\mathcal{D}$ is a finite set of facts. The grounding $\mathsf{ground}(\Pi)$ of a Datalog program Π is the set of all ground rules obtained by replacing variables in Π with constants from Dom. Given a dataset $\mathcal{D}$, $\mathsf{ADom}(\Pi, \mathcal{D})$ is the *active domain* of Π with respect to $\mathcal{D}$ and contains the set of constants from Π and $\mathcal{D}$. The grounding $\mathsf{ground}(\Pi, \mathcal{D})$ of a program Π with respect to a dataset $\mathcal{D}$ is the set of all ground rules obtained by replacing variables in Π with constants from $\mathsf{ADom}(\Pi, \mathcal{D})$. A predicate P is *extensional* if it occurs only in rule bodies and *intensional* if it occurs in the head of some rule. The extensional database (EDB) of Π consists of all extensional predicates of Π, while the intensional database (IDB) of Π, consists of all the intensional predicates of Π.

Example 2.1. Consider Example 1.4 again. Then, the active domain is $\{\text{john}, \text{anna}\}$ and the grounding $\mathsf{ground}(\Pi, \mathcal{D})$ of the rule produces the following rules:

$$\begin{aligned} \mathit{child}(\text{john}, \text{john}) &\leftarrow \mathit{father}(\text{john}, \text{john}) \\ \mathit{child}(\text{john}, \text{anna}) &\leftarrow \mathit{father}(\text{john}, \text{anna}) \\ \mathit{child}(\text{anna}, \text{john}) &\leftarrow \mathit{father}(\text{anna}, \text{john}) \\ \mathit{child}(\text{anna}, \text{anna}) &\leftarrow \mathit{father}(\text{anna}, \text{anna}) \end{aligned}$$

Semantics Datalog is defined via three different equivalent semantics, a proof-theoretic, an operational (fixpoint), and a model-theoretic one. In the following, we focus on the operational and the model-theoretic one and refer an interested reader to the literature for the proof-theoretic one [AHV95].

In the *model-theoretic semantic*, the idea is to interpret the program as a first order theory by associating each rule in Π with a first-order logical sentence. That is, for a rule of the form given in Equation (2.1), we associate the logical sentence

$$\forall X_1, \ldots, X_m (H \leftarrow B_1 \wedge \cdots \wedge B_k) \tag{2.2}$$

where $X_1, \ldots, X_m$ are the variables mentioned in the rule. Let Σ_Π be the conjunction of first-order logical sentences associated to the rules of Π. A model of Π is an interpretation $\mathfrak{I}$ (a function that specifies for each grounded atom whether it is true) that satisfies Σ_Π over a dataset $\mathcal{D}$. However, there can be infinitely many models of Π, so the decision taken by the designers of Datalog (and by the standard semantics in databases and logic programming) was to select an intended model as answer, which is a model that should not contain more facts than necessary. In Datalog, for each Datalog program Π and dataset $\mathcal{D}$, there exists a single unique minimal model of Σ_Π that extends $\mathcal{D}$. This unique minimal model is denoted by $\Pi(\mathcal{D})$.

Example 2.2. Consider Example 1.4 again. The rule in first-order logic would be:

$$\forall X, Y (\mathit{child}(Y, X) \leftarrow \mathit{father}(X, Y))$$

A model (but not a minimal model) for the given program and dataset is an interpretation that satisfies *child*(anna, john), *child*(john, anna), and *father*(john, anna). The unique minimal model satisfies only the facts *child*(anna, john) and *father*(john, anna)

In Datalog we often see that instead of using interpretations that assign a truth value to a ground atom, one considers a model as a dataset that contains all ground atoms that are satisfied by the interpretation. This allows one to work with sets that contain facts, i.e., write $P(\mathbf{s}) \in \mathcal{D}$ or if the unique minimal model entails the fact $P(\mathbf{s}) \in \Pi(\mathcal{D})$, instead of using an interpretation that satisfies a fact, i.e., write $\mathfrak{I} \models P(\mathbf{s})$ or $\Pi(\mathcal{D}) \models P(\mathbf{s})$, where $P(\mathbf{s})$ is some ground atom.

The *fixpoint semantics*, also called the operational semantics, relies on the *immediate consequence operator* T_Π that maps a dataset to a dataset. An atom A is an immediate consequence of a Datalog program Π and a dataset $\mathcal{D}$, if A is a ground EDB atom in $\mathcal{D}$, or $A \leftarrow B_1 \wedge \cdots \wedge B_k$ is a ground instance of a rule and $B_1, \ldots, B_k \in \mathcal{D}$. Then we can define T_Π as $A \in T_\Pi$ iff A is an immediate consequence of Π and $\mathcal{D}$. A *fixpoint* is now a dataset such that $T_\Pi(\mathcal{D}) = \mathcal{D}$. The *least fixpoint* for T_Π containing $\mathcal{D}$ is a fixpoint that is a subset of any other fixpoint for T_Π containing $\mathcal{D}$. For each Datalog program Π and dataset $\mathcal{D}$, there exists a minimal fixpoint containing $\mathcal{D}$, which is equal to the minimum model $\Pi(\mathcal{D})$. The operational semantics, hence, can be defined as a sequence of interpretations, where ω is a limit ordinal:

$$T_\Pi^0(\mathcal{D}) = \mathcal{D} \tag{2.3}$$

$$T_\Pi^{i+1}(\mathcal{D}) = T_\Pi(T_\Pi^i(\mathcal{D})) \tag{2.4}$$

$$\Pi(\mathcal{D}) = T_\Pi^\omega(\mathcal{D}) = \bigcup_{i \geq 0} T_\Pi^i(\mathcal{D}) \tag{2.5}$$

Note that T_Π is monotonic and hence $T_\Pi^i(\mathcal{D}) \subseteq T_\Pi^{i+1}(\mathcal{D})$. Also note that $T_\Pi^\omega(\mathcal{D})$, and thus $\Pi(\mathcal{D})$, can be obtained by applying T_Π finitely many times [GOPS12].

2.1.1 Stratified Datalog$^\neg$

Let us now extend Datalog with negation. For this, we extend the syntax of rules given in Equation (2.1) to take negated literals into account [AHV95, GHLZ13].

$$H \leftarrow B_1 \wedge \cdots \wedge B_k, \wedge \text{ not } B_{k+1} \wedge \cdots \wedge \text{ not } B_{k+m} \qquad \text{for } k \geq 0, m \geq 0 \tag{2.6}$$

We extend from Datalog without negation the notion of a rule (and thus for a program) to be safe by requiring in addition that each variable mentioned in the body of a negated literal appears also in a positive literal of the body of the rule.

In the following, we consider stratified negation, which provides tractable reasoning capabilities. The idea of this form of negation is to partition the program into subprograms, such that each subprogram only depends on negated predicates from the previous program. That is, one can interpret each subprogram as a semi-positive Datalog program (i.e., a program where negation is only applied to predicates of the EDB).

Definition 2.1.1. A *stratification* of a Datalog$^\neg$ program Π is an ordered partition of predicates in Π into strata $\Pi^1, \ldots, \Pi^n$ such that:

1. if $H \leftarrow \ldots \wedge B \wedge \ldots$ is a rule in Π, and H is in stratum Π_i while B is in stratum Π_j, then $i \geq j$
2. if $H \leftarrow \ldots \wedge \text{ not } B \wedge \ldots$ is a rule in Π, and H is in stratum Π_i while B is in stratum Π_j, then $i > j$

A Datalog$^\neg$ program that admits a stratification is called stratifiable.

In order to check, whether a program is stratifiable, one uses a dependency graph (see Definition 2.1.2), where we label an edge as $\neg$, if the body literal is negated. A Datalog$^\neg$ program Π is stratifiable iff its dependency graph G_Π has no cycle containing an edge labeled $\neg$.

Definition 2.1.2. The dependency graph G_Π of a program Π is the directed multi-graph with a vertex v_P for each predicate $P \in \Pi$ and an edge (v_P, v_Q) whenever there is a rule in Π mentioning P in the body and Q in the head.

With the definition of a dependency graph, we can now also formally define that a program Π is *recursive*, if G_Π has a cycle.

Example 2.3. Given a program Π with the following rules:

$$A \leftarrow B \qquad B \leftarrow A \qquad D \leftarrow A, \text{ not } C$$

Then the program has two strata: $\Pi^1 = \{A, B, C\}$ and $\Pi^2 = \{D\}$.

2.1.2 Datalog$^\pm$

The goal of Datalog$^\pm$ is to extend Datalog with existential quantification, with the truth constraint *false* in the head and with the equality predicate.

Existential rules extend our definition of a rule as follows

$$\exists \mathbf{X}\ H \leftarrow B_1 \wedge \cdots \wedge B_k, \wedge \text{ not } B_{k+1} \wedge \cdots \wedge \text{ not } B_{k+m} \qquad \text{for } k, m \geq 0 \tag{2.7}$$

where $\mathbf{X}$ is a tuple of variables which are mentioned in H but not in the rule body. We usually omit $\exists \mathbf{X}$ when writing a rule.

For selecting existential quantified variables, one usually differentiates between the *open world assumption* (OWA) where the assignments can use arbitrary constants from the domain Dom and the *closed world assumption* (CWA), where the assignments are restricted to only constants from the active domain ADom. In the context of knowledge graph reasoning, where we are interested in inventing new values, we usually consider only the open world assumption.

Rules with the truth constraint are given as follows

$$\bot \leftarrow B_1 \wedge \cdots \wedge B_k \qquad \text{for } k \geq 0 \tag{2.8}$$

As Datalog extended with existential quantification alone is already undecidable, certain syntactic restrictions have been studied to achieve the goal of preserving tractability [GOPS12]. We provide some of these restrictions in the following definitions [FKMP05, GOPS12, GLP14, GP15].

Definition 2.1.3. A rule is *guarded*, if for each rule there is at least one atom in the body that contains all the variables appearing in the body. A program is guarded if it contains a finite set of guarded rules.

Definition 2.1.4. A program is *linear* if each rule contains only one body atom.

Let us define a slightly modified dependency graph G'_Π of a program Π taking the relationship of variables into account. The dependency graph contains a vertex $v_{(P,i)}$ for each predicate P in Π and each position $i \in \{1, \ldots, n\}$, where n is the arity of P. There is an edge $v_{(P,i)}$ to $v_{(Q,j)}$ when there is a rule in Π mentioning $P(\mathbf{s})$ in the body and $Q(\mathbf{s}')$ in the head of the rule where the i-th position of $\mathbf{s}$ is a variable and the j-th position of $\mathbf{s}'$ shares the same variable. There is a special edge from any $v_{(P,i)}$ to $v_{(Q,j)}$ if the j-th position of $\mathbf{s}'$ is an existential variable and the i-th position of $\mathbf{s}$ shares the same variable with some position in $\mathbf{s}'$.

Definition 2.1.5. A program is *weakly acyclic*, if the dependency graph G'_Π of the program has no cycle containing a special edge.

Let us end the series of introduced restrictions by introducing the underlying fragment used in the Vadalog system, namely Warded Datalog$^\pm$. For this, we have to define harmless, harmful and dangerous variables. The i-th position of a predicate P is marked as affected, if there exists a rule with P in the head and this position being a variable, such that either the variable is existentially quantified, or all positions in the body mentioning this variable are marked as affected. Then, a variable in the body is *harmless* if at least one position mentioning the variable is not affected, *harmful* if it is not harmless and *dangerous* if it is harmful and the variable appears also in the head.

Definition 2.1.6. A program is *warded*, if for each rule there are no dangerous variables in the body or there exists an atom (the ward) in the body that contains all dangerous variables of the body and all variables that are shared between the ward and other body atoms are harmless.

Example 2.4. The following program is warded:

$$\begin{aligned} p(\underline{\mathrm{X}}, \mathrm{Z}, \underline{\mathrm{W}}) &\leftarrow s(\underline{\mathrm{X}}, \mathrm{Y}), r(\mathrm{Y}, \mathrm{Z}) \\ p(\underline{\mathrm{Z}}, \mathrm{W}, \underline{\mathrm{X}}) &\leftarrow p(\underline{\mathrm{X}}, \mathrm{Y}, \underline{\mathrm{Z}}), r(\mathrm{Y}, \mathrm{W}) \\ s(\underline{\mathrm{X}}, \mathrm{W}) &\leftarrow p(\underline{\mathrm{X}}, \mathrm{Y}, \underline{\mathrm{Z}}), r(\mathrm{Y}, \mathrm{W}) \end{aligned}$$

The affected positions are p[3] due to an existential variable in the first rule as well as p[1] and s[1] due to being part of only affected positions in the body. The underlined positions mark the affected positions in the program.

2.2 DatalogMTL

In this section we introduce DatalogMTL interpreted over the rational timeline and under the continuous semantics by building upon the concepts introduced for Datalog [BKK+17a, WCGKK19, TCWCGK21].

Time and Intervals. The timeline of DatalogMTL is defined over an ordered set of rational numbers $\mathbb{Q}$, where each time point t is an element of the timeline. An *interval* $\varrho = \langle \varrho^-, \varrho^+ \rangle$ is a subset of $\mathbb{Q}$, such that the endpoints $\varrho^-, \varrho^+ \in \mathbb{Q} \cup \{-\infty, \infty\}$, and for each $t \in \mathbb{Q}$ where $\varrho^- \leq t \leq \varrho^+$ we have $t \in \varrho$. The brackets $\langle$ and $\rangle$ are either round or square brackets and denote whether the endpoints of the interval are excluded (denoted by round brackets) or included (denoted by square brackets). An interval ϱ is *punctual* if it contains exactly one number, i.e., it is of form $[t, t]$, where we often just write t. An interval is *positive* if it does not contain negative numbers, and *bounded* if both endpoints are rational numbers. The *length* of an interval, denoted by $|\varrho|$ is ∞ if the interval is unbounded, otherwise it is defined as $\varrho^+ - \varrho^-$.

Example 2.5. The interval $(-\infty, 10]$ is unbounded, is left-open and right-closed and has a length of ∞. The interval $[0, 40)$ is bounded, positive, left-closed, and right open and has a length of 40. The interval $[5, 5]$ or 5 is bounded, positive, punctual, left-closed and right-closed and has a length of 0.

Syntax. The syntax of DatalogMTL extends Datalog (with constraints) in the definition of literals (which are also called metric atoms) and their positions in rules as well as extends facts with intervals.

Formally, a literal is defined by the following grammar:

$$M \ ::= \ \top \ | \ \bot \ | \ P(\mathbf{s}) \ | \ \diamondminus_\varrho M \ | \ \boxplus_\varrho M \ | \ \boxminus_\varrho M \ | \ M \mathcal{S}_\varrho M \ | \ M \mathcal{U}_\varrho M.$$

Intuitively, the $\diamondminus/\boxminus/\mathcal{S}$ operators state that M has to hold in the past, while the $\diamondplus/\boxplus/\mathcal{U}$ operators state that M has to hold in the future. In detail, $\diamondminus_\varrho M$ states that the literal holds at time t if the literal M holds at *some* past time point in ϱ relative to t, $\boxminus_\varrho$ if M holds at *all* past time points in ϱ relative to t, and $M' \mathcal{S}_\varrho M$ if M holds at some past time point t' in ϱ relative to t and M' holds *between* t and t'.

A (non-existential) *rule* is an expression of the form

$$M' \leftarrow M_1 \wedge \cdots \wedge M_k, \wedge \text{ not } M_{k+1} \wedge \cdots \wedge \text{ not } M_{k+m} \qquad \text{for } k, m \geq 0, \tag{2.9}$$

where each M_i is a literal and M' is a literal not mentioning $\diamondminus$, $\diamondplus$, $\mathcal{S}$, and $\mathcal{U}$. For stratified DatalogMTL$^\neg$, i.e., programs containing rules mentioning a negated literal, $\bot$

Table 2.1: Semantics of ground literals

$\mathfrak{I}, t \models \top$	for each t
$\mathfrak{I}, t \models \bot$	for no t
$\mathfrak{I}, t \models \diamondsuit_\varrho M$	iff $\mathfrak{I}, r \models M$ for some r with $t - r \in \varrho$
$\mathfrak{I}, t \models \boxplus_\varrho M$	iff $\mathfrak{I}, r \models M$ for some r with $r - t \in \varrho$
$\mathfrak{I}, t \models \boxminus_\varrho M$	iff $\mathfrak{I}, r \models M$ for all r with $t - r \in \varrho$
$\mathfrak{I}, t \models \blacksquare_\varrho M$	iff $\mathfrak{I}, r \models M$ for all r with $r - t \in \varrho$
$\mathfrak{I}, t \models M_1 \mathcal{S}_\varrho M_2$	iff $\mathfrak{I}, r \models M_2$ for some r with $t - r \in \varrho$ and $\mathfrak{I}, s \models M_1$ for all $s \in (t', t)$
$\mathfrak{I}, t \models M_1 \mathcal{U}_\varrho M_2$	iff $\mathfrak{I}, r \models M_2$ for some r with $r - t \in \varrho$ and $\mathfrak{I}, s \models M_1$ for all $s \in (t, r)$

is in addition disallowed in the rule head. A *fact* over an interval ϱ is an expression $P(\mathbf{s})@\varrho$, with $P(\mathbf{s})$ a ground atom. The other definitions (e.g., program, grounding, dataset, stratification and so on) are defined analogously to Datalog.

Example 2.6. The following program is a DatalogMTL program expressing that a person X is employed at a company Y between their join and leave events:

$$\mathit{employed}(\mathrm{X}, \mathrm{Y}) \leftarrow (\diamondsuit_{[0,\infty)} \mathit{join}(\mathrm{X}, \mathrm{Y})) \; \mathcal{U}_{[0,\infty)} \; \mathit{leave}(\mathrm{X}, \mathrm{Y})$$

In detail, assume, that the employee e joins a company c at time 5 and leaves at time 10. Thus we have that $\mathit{join}(\mathrm{e}, \mathrm{c})@[5, 5]$, and $\mathit{leave}(\mathrm{e}, \mathrm{c})@[10, 10]$ hold. When we apply $\diamondsuit$, we extend the resulting fact to the infinite future, i.e., one derives an intermediary result that holds in the interval $[5, \infty)$. Then, with the application of $\mathcal{U}$, we combine this resulting interval with the *leave* fact. For this we consider the last line of Table 2.1, which has two conditions, one before "and", and one after it. The first condition results in an interval $(-\infty, 10]$ and the second condition restricts this further to $[5, 10]$, i.e., resulting in $\mathit{employed}(\mathrm{e}, \mathrm{c})@[5, 10]$.

Semantics. The semantic definition extends the notion of the model-theoretic semantic of Datalog to DatalogMTL taking the temporal domain into account. An *interpretation* $\mathfrak{I}$ is a function which specifies for each ground atom $P(\mathbf{s})$ and each time point $t \in \mathbb{Q}$, whether $P(\mathbf{s})$ is *satisfied* at t, in which case we write $\mathfrak{I}, t \models P(\mathbf{s})$. This notion of satisfiability extends to ground literals as given in Table 2.1. Interpretation $\mathfrak{I}$ satisfies a fact $P(\mathbf{s})@\varrho$, written $\mathfrak{I} \models P(\mathbf{s})@\varrho$, if $\mathfrak{I}, t \models P(\mathbf{s})$ for all $t \in \varrho$ and a dataset $\mathcal{D}$ if it satisfies all facts in $\mathcal{D}$. Interpretation $\mathfrak{I}$ satisfies not M, written $\mathfrak{I}, t \models \text{not } M$ if $\mathfrak{I}, t \not\models M$. Interpretation $\mathfrak{I}$ satisfies a ground rule at t if, whenever $\mathfrak{I}$ satisfies each body literal at time t, then $\mathfrak{I}$ also satisfies the head of the rule at t; and a rule if it satisfies each possible grounding of the rule. Interpretation $\mathfrak{I}$ is a *model* of a program Π if it satisfies each rule in Π. A program Π and a dataset $\mathcal{D}$ *entail* a fact $P(\mathbf{s})@\varrho$, if each model of both Π and

Algorithm 2.1: Materialization procedure for DatalogMTL

Input: A DatalogMTL program Π and a dataset $\mathcal{D}$
Output: A dataset $\mathcal{D}'$ representing materialization of Π and $\mathcal{D}$, or 'inconst' if Π and $\mathcal{D}$ are inconsistent

$\mathcal{D}' \leftarrow \mathcal{D}$;
repeat
  for *each* $r \in \mathsf{ground}(\Pi, \mathcal{D}')$ *and a (maximal) interval* ϱ *at which* $\mathcal{D}'$ *entails the body of* r **do**
    $H \leftarrow$ head of r;
    if H *mentions* $\bot$ **then**
      return inconst;
    $\mathcal{D}' \leftarrow \mathcal{D}' \cup \{H\}$;
  Coalesce facts in $\mathcal{D}'$;
until *no change to* $\mathcal{D}'$;
return $\mathcal{D}'$

$\mathcal{D}$ is a model of $P(\mathbf{s})@\varrho$ and are *consistent* if they have a model (in case $\bot$ is used in the head). If a program Π and a dataset $\mathcal{D}$ have a model, then they also have a unique minimal model $\Pi(\mathcal{D})$ (also called the canonical interpretation).

We want to remark that in Datalog one often considers a model as a dataset instead of using the notion of interpretations. Thus, sometimes we abuse the notion of interpretations and just simply write $M@\varrho \in \mathfrak{I}$ if $\mathfrak{I} \models M@\varrho$.

The notions of the fixpoint-operation are defined analogously to Datalog for consistent DatalogMTL programs with the only difference that the unique minimal model is not always obtained in finite time but requires at most ω_1 (the first uncountable ordinal) applications.

A possible materialization algorithm for computing the least fixpoint for DatalogMTL programs is given in Algorithm 2.1, taking also inconsistent programs into considerations, where coalescing of facts means that facts with overlapping or adjacent intervals ϱ_1, ϱ_2 are recursively replaced with a fact over an interval $\varrho_1 \cup \varrho_2$. Note that this algorithm only terminates, if the program is inconsistent or the least-fixpoint computation is obtained in finite time.

Normal form. We usually assume that programs are given in *temporal normal form* [WCGKK19] where each rule satisfies additional requirements (e.g., no nesting of temporal operators) resulting in following allowed rules:

$$\bot \leftarrow P_1(\mathbf{s}_1) \wedge \cdots \wedge P_n(\mathbf{s}_n) \qquad (n \geq 0) \qquad (2.10)$$

$$P_0(\mathbf{s}_0) \leftarrow P_1(\mathbf{s}_1) \wedge \cdots \wedge P_n(\mathbf{s}_n) \qquad (n \geq 0) \qquad (2.11)$$

$$P_0(\mathbf{s}_0) \leftarrow P_1(\mathbf{s}_1)\ \mathcal{S}_\varrho\ P_2(\mathbf{s}_2) \qquad (2.12)$$

$$P_0(\mathbf{s}_0) \leftarrow P_1(\mathbf{s}_1)\ \mathcal{U}_\varrho\ P_2(\mathbf{s}_2) \tag{2.13}$$
$$P_0(\mathbf{s}_0) \leftarrow \boxminus_\varrho P_1(\mathbf{s}_1) \tag{2.14}$$
$$P_0(\mathbf{s}_0) \leftarrow \boxplus_\varrho P_1(\mathbf{s}_1) \tag{2.15}$$
$$P_0(\mathbf{s}_0) \leftarrow \diamondminus_\varrho P_2(\mathbf{s}_1) \tag{2.16}$$
$$P_0(\mathbf{s}_0) \leftarrow \diamondplus_\varrho P_2(\mathbf{s}_1) \tag{2.17}$$

where ϱ is non-empty and non-negative and $\varrho^- = 0$ for all unbounded intervals. We want to note, that in literature one finds different representations of the normal form, for example, by allowing also $\top$ in addition to P_1 of 2.12 and 2.13, one would cover the rules (2.14), and (2.16), yet making it harder to define some fragments such as the one we define next.

Definition 2.2.1. We call a program forward propagating, or in short DatalogMTL^{FP} if it contains only rules of the form (2.11), (2.14), and (2.16) [WKCG19].

2.3 Temporal Aggregation

In database theory, various kinds of temporal aggregation have been studied that differ in how the values are grouped together along the timeline. For the concepts, we mainly follow the given descriptions on the Encyclopedia of Database Systems [GBJ18].

One distinguishes between three relevant ways to compute temporal aggregations, the instantaneous temporal aggregation (in short ITA), the moving-window temporal aggregation (MWTA) and the span temporal aggregation (STA). Furthermore, there is the multi-dimensional temporal aggregation (MDTA) which relies on one of the other aggregation types.

Example 2.7. Let us consider the following table, which tracks the interval of each person owning a company and whether the person is male (m) or female (f). We will use this table in the following to explore the different aggregation types.

Id	m/f	Interval
1	m	[1980,1984]
2	f	[1984,1988]
3	m	[1992,1998]
4	f	[2012,2014]
5	f	[2016,2022]

The ITA partitions the timeline into time instants and each time instant t is associated with its own aggregation group. Each aggregation group contains all facts that are valid in t, and hence each aggregation group produces a single aggregation value at each time instant t. Finally, consecutive time instants with the same value are coalesced into constant intervals, i.e., maximal intervals over which the aggregation result remains constant.

The MWTA uses a time-window for each time instant t to define the values being part in an aggregation group. That is, it extends ITA, where the window can be seen as $[t, t]$, with either a single offset $w \geq 0$ yielding a window $[t - w, t]$, or a pair of offsets $w \geq 0$ and $w' \geq 0$ yielding a window $[t - w, t + w']$ for determining the value of an aggregation group. The remaining process is equal to the ITA.

Example 2.8. A typical question for ITA is to ask for the number of people owning a company. For MWTA we would extend the question by asking for the number of people owning a company in the last 10 years (i.e., a window of $(t - 10, t]$). An answer regarding the dataset is as follows:

ITA		MWTA	
Count	Interval	Count	Interval
1	[1980,1984)	1	[1980,1984)
2	[1984,1984]	2	[1984,1992)
1	(1984,1988]	3	[1992,1994)
0	(1988,1992)	2	[1994,1998)
1	[1992,1998]	1	[1998,2008)
0	(1998,2012)	0	[2008,2012)
1	[2012,2014]	1	[2012,2016)
0	(2014,2018)	2	[2016,2024)
1	[2016,2022]	1	[2024,2032)

The STA partitions the timeline into predefined intervals forming an aggregation group. Each fact overlapping with the interval is associated with this aggregation group. One usually considers for the selection of intervals regular time spans such as weeks, months or years.

Example 2.9. A typical question for STA is to ask for the number of people owning a company per decade. An answer regarding the dataset is as follows:

Count	Interval
2	[1980,1989)
1	[1989,1999)
0	[2000,2009)
2	[2010,2019)
1	[2020,2029)

Finally, the MDTA decouples the result from the aggregation group allowing to specify parts of the output that is independent of the aggregation.

Example 2.10. A typical question for MDTA is to ask for the number of male and female people owning a company per decade. An answer regarding the dataset is as

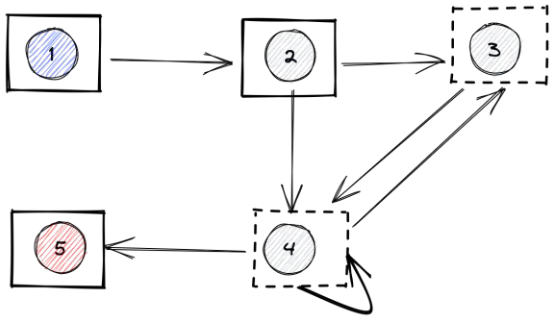


Figure 2.1: Vadalog system

follows:

Count M	Count F	Interval
1	1	[1980,1989)
1	0	[1989,1999)
0	0	[2000,2009)
0	1	[2010,2019)
0	1	[2020,2029)

2.4 Vadalog

As previously mentioned, Vadalog is built on Warded Datalog$^{\pm}$. In this section, we summarize the two main architectural components of the Vadalog system, namely the pipeline architecture and its execution model, as well as termination strategies and cycles [BBGS20].

Pipeline and Execution Model. Vadalog follows the pipes-and-filters style, where filters transform data between their input and output flow and are connected by pipes (buffers). For the construction of this pipeline, Vadalog applies a logical optimizer that rewrites certain rules (e.g., removing multiple rule heads) and applies general optimizations, a logic compiler that transform the optimized rules to a plan consisting of pipes and filters and a query compiler that instantiates the plan with its underlying data sources. Then, the reasoning is triggered by the sinks of the plan, which invoke their parents to fetch new facts until no new facts are derived. In case there are multiple parents for a filter, Vadalog supports multiple routing options to select the next parent to pull data from (e.g., round robin or shortest EDB path) [BBGS20].

Termination Strategies and Cycles. The idea of the termination strategies is to guarantee termination of an in general infinite reasoning process. That is, one stops the propagation of existential variables in such a way that all relevant facts for the reasoning process are generated (i.e., one does not lose facts). One such termination strategy is the

isomorphism strategy, where facts that are isomorphic to an existing fact are not further propagated. Note that two facts are isomorphic, when they have the same predicate name, in the same positions the same constants and there exists a bijection between the assigned values for existential variables. The presence of non-terminating sequences and thus the application of termination strategies appear by the presence of cycles in the program. In such cases, one equips the filters that are part of a cycle with a termination strategy. In case the termination criteria is met (e.g., an isomorphism is detected), the pulled fact from the parent is discarded [BBGS20].

Figure 2.1 visualizes the architecture. Squares represent filters, and dotted squares denote filters equipped with a termination wrapper. The arrows denote that X reads from Y, e.g., 2 reads data from 1. The blue nodes mark input predicates (EDBs) and the red nodes mark output predicates (sinks).

Example 2.11. Consider Figure 2.1 again. As discussed, in the pipes-and-filters architecture, nodes pull from their parent. For example, we see that node 2 pulls from node 1, which represents the input. In turn, node 3 pulls from nodes 2 and 4. Note that this pulling is quite interesting as it is part of a cycle. This requires the termination strategy (discussed in a previous paragraph) to prevent infinite loops. The other nodes follow the same pattern. In addition, nodes may filter results that flow through them, making up in total the pipes-and-filters architecture.

2.5 Related Work

This section summarizes the current state of research in the several areas to which this thesis contributes. We begin by providing an overview over discussions of temporal reasoning in Datalog and similar languages, with an emphasis on the current research around DatalogMTL, in Section 2.5.1. Then, we focus in Section 2.5.2 on summarizing the discussions regarding recursive aggregation in Datalog. Finally, we discuss in Section 2.5.3 the landscape of useable benchmarks for evaluating temporal programs.

2.5.1 Temporal Reasoning

First temporal extensions to Datalog were suggested already in the 1980s. Most approaches can be grouped into two types, one focusing on implementing temporal constructs via arithmetic operations, e.g., by applying the +1 function to model different discrete temporal units (Datalog_{1S} [Cho90, RKG+18, Zan12]), the other including operators from temporal logic such as the always and eventually operator from LTL or CTL into Datalog (Templog [AM89], DatalogLite [GGV02]). Templog as well as Datalog_{1S} are expressively equivalent and can be seen as subsets of DatalogMTL. There is also a temporal extension with the Halpern-Shoham logic of intervals [KPP+16].

Newest developments are based on MTL [BKK+17a, WCGKK19, KRWZ18], an extension of LTL to enrich the expressive power of Datalog programs. DatalogMTL has been

Table 2.2: Data complexity classes of DatalogMTL fragments with continuous semantics

<table>
<tr><th></th><th colspan="2">Rational Timeline</th><th colspan="2">Integer Timeline</th></tr>
<tr><th></th><th>First-order</th><th>Prop.</th><th>First-order</th><th>Prop.</th></tr>
<tr><td>strat. DatalogMTL$^{\neg}$</td><td rowspan="3" colspan="2">PSPACE-c</td><td rowspan="2">PSPACE-c</td><td>unknown</td></tr>
<tr><td>DatalogMTL</td><td rowspan="5">NC^1-c</td></tr>
<tr><td>DatalogMTL$_{\text{lin}}$</td></tr>
<tr><td>DatalogMTL$_{\text{core}}$</td><td colspan="2"></td><td>NL-h</td></tr>
<tr><td>DatalogMTL$^{\boxminus}_{\text{lin}}$</td><td rowspan="2">P-hard</td><td>PSPACE-c</td></tr>
<tr><td>DatalogMTL$^{\boxminus}_{\text{core}}$</td><td colspan="2"></td><td>NC^1-h</td></tr>
<tr><td>DatalogMTL$^{\diamondminus}_{\text{lin}}$</td><td colspan="2">NL-c</td><td>NL-c</td></tr>
<tr><td>DatalogMTL$^{\diamondminus}_{\text{core}}$</td><td colspan="2">TC^0-c</td><td colspan="2">in AC$^0[k]$ for $k \in \mathbb{N}$, not in AC^0</td></tr>
</table>

studied for the continuous, which also our work concentrates on, as well as for the pointwise [KRWZ18] semantics, where the timeline consists only of time points explicitly mentioned in a dataset. In the following, we only focus on the continuous semantics which provides a more natural understanding of temporal operators [Rey16].

Early work on DatalogMTL primarily focused on the complexity results of fact entailment, i.e., if a program and a dataset entail a given fact (and consistency checking, i.e., if a given program and a dataset have a model) [BKR+18, WCGKK19, TCWCGK21, WTCKCG21]. This includes sub-fragments that restrict the operators [WCGKK20b], that restrict the time points to the integer domain [WCGKK20a], or that consider only finitely materializable programs [WZCG21] as well as extensions such as stratified negation [TCWCGK21] or negation under stable model semantics [WTCKCG21]. An overview of the mainly derived complexity classes is given in Table 2.2. The core fragment is defined as rules of the form $B \leftarrow A$ or $\bot \leftarrow A_1 \wedge A_2$, and the linear fragment as rules of the form $B \leftarrow A_1 \wedge \ldots \wedge A_n$ with at most one A_i being an IDB literal or of form $\bot \leftarrow A_1 \wedge A_2$. DatalogMTL$^{\diamondminus}$ and DatalogMTL$^{\boxminus}$ denote that the mentioned temporal operator (i.e., $\diamondminus$ or $\boxminus$) is the only temporal operator allowed to be used in rules.

While most of the early work for DatalogMTL purely consisted of complexity results, there is a proposal for a non-recursive fragment [BKK+17a], as well as an algorithm for stream reasoning [WKCG19] for the forward propagating fragment. During our work on the thesis, additional practical results were presented, including a DatalogMTL reasoner, called MeTeoR [WHWCG22, WWCG22], which uses a mixture of materialization (i.e., least fixpoint computation) and automaton-based reasoning to check whether a given fact can be entailed.

Closely related to Datalog are temporal extensions to ASP as well as description logic. Most prominent is LARS [BDEF15], a temporal stream reasoning framework focusing on finite streams by extending propositional logic. It supports the usage of a window operator that returns a sub-stream containing only n time points. In comparison with DatalogMTL it offers a model-centric perspective (instead of a query-centric). Further, there is also extension of ASP with metric temporal logic [CDSS22] over the integer

timeline. Metric temporal logic has also been considered as extensions to description logics [GJO16].

2.5.2 Aggregation

Similarly, aggregation in Datalog has been discussed for many years. The biggest issue regarding aggregation is that the standard Datalog fixpoint semantics is defined over monotonic transformations (w.r.t. set containment) and aggregation breaks this requirement. Earlier solutions [RS92, ZAO93] use non-deterministic choice constructs, partial orders that are more powerful than set containment, or an infinite level of stratification. These approaches were of limited generality and also required to resort to sophisticated compilers to detect monotonic programs [MSZ13]. Datalog^{FS} [MSZ13] is the first approach that uses continuous aggregate functions to support monotonic counts, sums over positive values, and extrema aggregates (min,max). Shkapsky et al. [SYZ15] provided first practical algorithms for monotonic aggregation by introducing contributor and group-by terms, which we use in the following as foundations for our approach. While their work focused on optimizing aggregation for plain Datalog (i.e., a single time point), our approach aims at building monotonic aggregates for all time points efficiently, which requires specific handling of fact validity intervals, and dealing with time axis aggregation. Recent work studied the definition of min and max over limit Datalog [KGKH20], but only considers restricted use of sum and count by means of a sorted list of facts. Wang et al. [WZW$^+$20] studied techniques to convert non-monotonic aggregates to monotonic ones. The newest findings study aggregations by using semirings [WKN$^+$22, KNP$^+$22] while keeping the standard least fixpoint operation in place. Similar ideas can be found in a recent extension of the ASP reasoner LARS [EK20], where a formula can be evaluated as an algebraic expression over a semi-ring, allowing to compute all aggregates bounded by the semi-ring along the time-axis, e.g., they support to count the number of time points at which an expression holds.

Regarding implementation, in temporal databases a lot of approaches depend on tree structures to handle temporal aggregation. For example, one of the first algorithms uses an aggregation tree [KS95]. This tree contains as leaf-nodes the query results of constant intervals (i.e., the smallest intervals where in-between no tuples end or start, so that the aggregation value cannot change). In the worst case, this tree is not balanced, yielding a linked list of values. In addition, different suggestions were proposed that use different tree structures, such as AVL-Trees [BGJ06] that are used to keep track of the end-points of the tuples and add the result of an interval, if an endpoint of one of the added tuples is reached, or MVSB-Trees [ZMT$^+$01] (Multi Version SB-trees), based on SB-Trees [YW03] and Multiversion B-Trees [BGO$^+$96] that focus on a particular interval and key-range. Newest findings are based on a temporal index [KMV$^+$13], i.e., a structure containing two data structures, (i) an event list that marks which entries are activated or invalidated and (ii) a version map that keeps track of which events were visible at a certain version (time) in the database. In addition, it uses a checkpoint table to store which rows are active after a certain point in time to avoid recalculation for each

query. Calculating an aggregation requires a sequential reading of the list, applying the activation and invalidation operation to the current result (or in cases of non-cumulative aggregates storing the top k values plus in addition the other values in unsorted lists for accessing the next value efficiently). One such extension that improves the performance is a sweeping-based algorithm [PH17], adding support for fixed intervals and additional operations for constant intervals. In order to support also application time, Kaufmann et al. [KFM+15] extended the structure with an application time event map storing the valid events for the application times at a certain transaction time and a set of visibility bitmaps indicating whether a row is visible (a checkpoint for the application time).

2.5.3 Benchmarks

Recent work on temporal benchmarks for databases focused on extending the widely used TPC-H benchmark with temporal dimensions. Al-Kateb et al. [ACGR12] extended the tables with temporal columns and use the original queries and generator as part of the workload. In comparison, Kaufmann et al. [KFM+13] derive valid time from existing information. They introduce "time travel", audit (ignoring or providing constraints for the time-dimensions for a single key), range-timeslice, and bitemporal queries. These benchmarks are highly optimized for the relational model and do not focus on graph-based structures. In addition, the benchmark generator and the benchmark dataset have not been published and hence cannot be reused.

Similarly, time-series benchmarks, such as TSBS [Tim] or TS-Benchmark [HQC+21] are targeting time-series databases that focus on aggregation, range and rolling window queries, but not on joining multiple tables. TS-Benchmark uses a generative adversarial network and random-walk to generate synthetic data based on real-world datasets.

The semantic web community has worked on several benchmarks. SRBench [ZPCC12] is a benchmark for streaming RDF/SPRAQL queries based on linked open data cloud using LinkedSensorData interlinked with GeoNames and DBPedia. It contains 17 queries, including grouping per hour, per sliding window, checking for missing data over a predefined timeframe, aggregation, and hierarchical queries. However, this benchmark contains no queries that follow a cyclic pattern. The Linear Road benchmark [ACG+04] focuses on Data Stream Management Systems, i.e., on relational data and not on a graph-based structure. It supports window-based queries, aggregations, various kinds of complex joins, e.g., theta joins or self-joins. SLUBM [NS13] is another benchmark that extends the university ontology LUBM with timestamps by choosing a semester granularity and forcing semester-specific data to expire. For this, they introduce a "time-to-last" value to represent the expiration time of the conclusions and reuse 14 LUBM queries. CityBench [AGM15] is another benchmark that focuses on spatial and temporal data. It supports similar queries as SRBench but focuses on real city data.

Apart from that, Timescales [Ulu19] is a benchmark generator for MTL monitoring tools that is able to test ten different properties expressible by MTL (e.g., an event P does not/always occur a number of pre-specified time units after/before another event Q or

between two events Q and R). Hoxha et al. [HAF14] propose a benchmark for automotive systems that is based on MTL queries and focuses on checking the operations of gear changes, reached vehicle velocity, and engine speed (e.g., there should not be another gear change within x seconds). In a recent work, Wang et al. [WHWCG22] extended the LUBM benchmark with temporal rules and data, making it possible to benchmark DatalogMTL, yet missing key components such as aggregation.

CHAPTER 3

Functionality – Temporal Aggregation and Existential Quantification

In this chapter, we introduce two key capabilities for knowledge graph reasoning missing in DatalogMTL that are similar to the extensions that transform Datalog into Vadalog. The first contribution extends DatalogMTL with monotonic aggregation. This work is based on the paper [BNS21a] and is presented in Section 3.1. The second contribution extends DatalogMTL with existential quantification and is based on the paper [LNSW23], where our main contribution was on the applied aspects of the paper. We introduce the key definitions and an overview of the theoretical results as well as a novel algorithm in Section 3.2 and present the evaluation in a following chapter. Finally, we summarize the chapter in Section 3.3.

3.1 Monotonic Aggregation for DatalogMTL

The aggregation of temporal data is an essential capability in many data science workflows as it allows to summarize data along the timeline. Typical scenarios are Internet of Things (IoT) applications, where temporal data from heterogeneous sources (e.g., sensor streams) is aggregated [BGZ+20], state-of-the-art security information and event management systems (SIEM) with time-based alerts on aggregated log events [YOO+13], or the analysis of economic phenomena, where aggregation over time series is traditionally used for many settings such as the comparison of seasonal data or the detection of economic trends.

Especially for the economic domain, it has been shown that a Datalog-based knowledge graph with monotonic aggregation (i.e., aggregation that is only increasing/decreasing

with new values) provides sufficient expressive power and scalability in many applications of the financial realm, including company ownership, fraud detection or prevention of potential takeovers [BMNS20]. Consequently, it is logical to build on the work of non-temporal aggregation and consider a number of paradigmatic use cases of temporal reasoning and aggregations that have emerged as a basis for further analysis.

- **UC1: Revenue Calculation**. Shareholders are interested in the revenue of a company per week/month/year or over the complete lifespan.
- **UC2: Number of Trades**. Financial analysts are interested in the number of trades in the last hour/day.
- **UC3: Company Ownership/Shares**. Detecting hidden company ownerships is important to study and, if the case, prevent company takeovers. Having this information not only for a single point in time would provide deeper insights on the takeover determinants and would improve the prediction accuracy.
- **UC4: Change of Control**. Analysts wish to analyze ownership structures over time, e.g., monotonically increasing or decreasing shareholding.

In summary, these four use cases highlight different aggregation scenarios. We consider the first three use cases as *time-point aggregation* as the aggregation is applied on single time points, or time points in an interval. That is, UC1 requires intervals of fixed periods (which may vary in size, e.g., a month has 28 to 31 days), UC2 has a moving window of fixed size (e.g., an hour, a week) and UC3 requires to aggregate temporal information recursively over structures (e.g., over paths of arbitrary length). Differently from the previous cases, UC4 requires aggregating potentially along the entire time axis and is not bounded by any pre-determined interval; thus, we talk about *time-axis aggregation*.

In this section, we investigate monotonic aggregation in DatalogMTL. First, we derive in Section 3.1.1 a set of requirements for temporal aggregation which focus on general desiderata for declarative languages. Then, in Section 3.1.2 we focus on time point aggregation, discussing use cases UC1 to UC3 and provide the syntax and semantic of time point aggregation. Thereby, we introduce an additional granularity operator for DatalogMTL for covering spanning time aggregation and introduce an algorithm that take into account the intervals in the computation of the monotonic aggregates. We finish in Section 3.1.3 with a discussion on time-axis aggregation. In detail, we study the syntax and semantics of detecting monotonic trends and provide an efficient algorithm for computing those trends.

3.1.1 Requirements

In this section, we analyze the *requirements of temporal reasoning and aggregation* in DatalogMTL. We first consider general requirements that are desirable in a declarative AI solution and then instantiate them into specific requirements for temporal aggregation:

$$tradeInterval(\mathrm{U}, \mathrm{Id}, \mathrm{T}, \mathrm{T}+3600, 1, 0) \leftarrow trade(\mathrm{U}, \mathrm{Id}, \mathrm{T}) \quad (1)$$

$$iPoint(\mathrm{X}, \mathrm{X}_b), iPoint(\mathrm{Y}, \mathrm{Y}_b) \leftarrow tradeInterval(\mathrm{U}, \mathrm{Id}, \mathrm{X}, \mathrm{Y}, \mathrm{X}_b, \mathrm{Y}_b) \quad (2)$$

$$int'(0, mmin(\mathrm{T}), 1, \mathrm{C}) \leftarrow iPoint(\mathrm{T}, \mathrm{C}) \quad (3a)$$

$$int(\mathrm{X}, \mathrm{Y}, 1, 1) \leftarrow int'(\mathrm{X}, \mathrm{Y}, 1, 0) \quad (3b)$$

$$int(\mathrm{X}, \mathrm{Y}, 1, 0) \leftarrow int'(\mathrm{X}, \mathrm{Y}, 1, 1) \quad (3c)$$

$$int'(\mathrm{T}_p, mmin(\mathrm{T}), 0, \mathrm{C}) \leftarrow int(\_, \mathrm{T}_p, \_, 1), iPoint(\mathrm{T}, \mathrm{C}), \mathrm{T} > \mathrm{T}_p \quad (4a)$$

$$int'(\mathrm{T}_p, mmin(\mathrm{T}), 1, \mathrm{C}) \leftarrow int(\_, \mathrm{T}_p, \_, 0), iPoint(\mathrm{T}, \mathrm{C}), \mathrm{T} > \mathrm{T}_p \quad (4b)$$

$$int(\mathrm{X}, \mathrm{Y}, 0, 1) \leftarrow int'(\mathrm{X}, \mathrm{Y}, 0, 0) \quad (4c)$$

$$int(\mathrm{X}, \mathrm{Y}, 0, 0) \leftarrow int'(\mathrm{X}, \mathrm{Y}, 0, 1) \quad (4d)$$

$$V(I, \mathrm{Id}, \mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c) \leftarrow int(\mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c), tradeInterval(\mathrm{U}, \mathrm{Id}, \mathrm{S}_t, \mathrm{E}_t, \mathrm{S}_c, \mathrm{E}_c), (\mathrm{X} < \mathrm{E}_t), (\mathrm{Y} > \mathrm{S}_t) \quad (5a)$$

$$V(\mathrm{U}, \mathrm{Id}, \mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c) \leftarrow int(\mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c), tradeInterval(\mathrm{U}, \mathrm{Id}, \mathrm{S}_t, \mathrm{E}_t, \mathrm{S}_c, \mathrm{E}_c), (\mathrm{X} = \mathrm{E}_t \wedge \mathrm{X}_c = 1 \wedge \mathrm{E}_c = 1), (\mathrm{Y} > \mathrm{S}_t) \quad (5b)$$

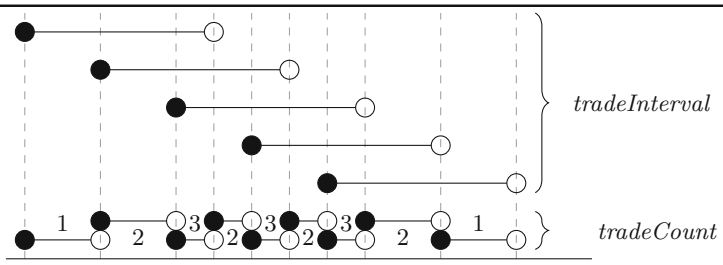
$$V(\mathrm{U}, \mathrm{Id}, \mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c) \leftarrow int(\mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c), tradeInterval(\mathrm{U}, \mathrm{Id}, \mathrm{S}_t, \mathrm{E}_t, \mathrm{S}_c, \mathrm{E}_c), (\mathrm{X} < \mathrm{E}_t), (\mathrm{Y} = \mathrm{S}_t \wedge \mathrm{Y}_c = 1 \wedge \mathrm{S}_c = 1) \quad (5c)$$

$$V(\mathrm{U}, \mathrm{Id}, \mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c) \leftarrow int(\mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c), tradeInterval(\mathrm{U}, \mathrm{Id}, \mathrm{S}_t, \mathrm{E}_t, \mathrm{S}_c, \mathrm{E}_c), (\mathrm{X} = \mathrm{E}_t \wedge \mathrm{X}_c = 1 \wedge \mathrm{E}_c = 1), (\mathrm{Y} = \mathrm{S}_t \wedge \mathrm{Y}_c = 1 \wedge \mathrm{S}_c = 1) \quad (5d)$$

$$tradeCount(\mathrm{U}, mcount(\mathrm{Id}), \mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c) \leftarrow V(\mathrm{U}, \mathrm{Id}, \mathrm{X}, \mathrm{Y}, \mathrm{X}_c, \mathrm{Y}_c) \quad (6)$$

In a first step, for all four aggregation types, we calculate all interval points at which the query answer may change in Rules 1-4. Rule 1 extends the *trade* facts to one-hour intervals *tradeInterval* (i.e., the desired time of the query) and defines that the intervals are left-closed and right-open (represented by 1 resp. 0). Rule 2 extracts all relevant interval points *iPoint*. Rules 3 and 4 create intervals *int* between all extracted points and assign open and closed properties. We continue in Rule 5 with detecting overlaps between *tradeInterval* and *int* and assigning the values of *tradeInterval* to the overlapping intervals *Int*. In particular, this rule covers overlapping checks of all combinations of open and close intervals. Finally, Rule 6 calculates the number of trades per interval and user (so we are grouping per user and time-interval). In order to support other aggregation types, we have to adapt Rule 6, where the aggregation has to be changed accordingly.

Figure 3.1: UC2 in non-temporal Datalog



This example has five *trade* events already extended to one-hour intervals. The five top-most intervals denote all *tradeInterval* predicates. Vertical, dashed lines show all *iPoint* predicates. The intervals at the bottom refer to the final result, that is, the *tradeCount* predicates.

Figure 3.2: Example for number of trades per hour

RQ1: Declarative. While the aggregation over temporal data can be simulated to a certain extend in Datalog, managing such temporal properties calls for complex interval checking logic and arithmetic. This easily leads to an error-prone and labor-intensive procedural approach which has to be repeated for each desired aggregation. Declarative temporal operators that specify *what* should be done instead of *how* it should be done avoid such pitfalls.

Example 3.1. A baseline implementation for computing the number of trades for the last hour would first extend time-point intervals to an hour-long intervals, then split the intervals into independent "counting" intervals—at each start or end of some interval in the data the count changes, i.e., either a trade is added or removed from the data—and finally count the number of entries per interval. We have provided such a sketch in Figure 3.1 and an example of this sketch in Figure 3.2.

RQ2: Implicit Time. Explicit time in rules provides the user the possibility to access and modify temporal properties. This increases the degree of freedom to handle temporal operations and thus requires certain semantic restrictions for the chosen operations to block arbitrary rule behavior. Implicit time handling, such as in temporal logic or DatalogMTL, does not have such issues, allowing for a fully declarative, composable solution.

Example 3.2. For example, if time is explicit part of tuples and manipulated using arithmetic, a user can add arbitrary arithmetic operations to the start and end points of a rule as seen in the following examples:

$$R(\mathrm{S}+5,\mathrm{E}+5) \leftarrow P(\mathrm{S},\mathrm{E}) \qquad R(\mathrm{S}*\mathrm{S},\mathrm{E}+5) \leftarrow P(\mathrm{S},\mathrm{E})$$

With implicit time in DatalogMTL, the information is handled via temporal operators, which expresses the first rule as follows, while the second rule is not possible to be

expressed in DatalogMTL:

$$R \leftarrow \diamondminus_{[5,5]} P$$

RQ3: Optimizability. Temporal data often stays the same for a longer interval (e.g., the president of a country is elected for multiple years), while temporal operations are defined per time point (e.g., counting the number of shares of a company). While declarative operators and implicit time already lead to a certain degree of optimizability, any operators defined should take this fact into account and exploit the evaluation over intervals for yielding an optimization evaluation of the operator.

Example 3.3. For example, if we have two facts $A(3)@[0,4]$ and $A(2)@[2,5]$, we only have changes in the aggregation values at time points 0, 2, 4 and 5. That is, we require in total an aggregation in the range $[0,2)$, $[2,4]$ and $(4,5]$ with the results 3, 5, and 2 respectively.

We also need support for fundamental types of temporal aggregation, as e.g., required by the archetypal use cases discussed in the beginning of Section 3.1:

RQ4: Moving window Temporal Aggregation. The ability to aggregate over a fixed time window, e.g., aggregate all values across the last hour. This should at least support aggregates min, max, count, sum.

This allows the scenarios of UC2 and UC3.

RQ5: Span Temporal Aggregation. The ability to aggregate over a fixed interval, e.g., aggregate all values from the month of April 2021. This should at least support aggregates min, max, count, sum.

This allows the scenarios of UC1 and UC3.

RQ6: Time Axis Aggregation. The ability to aggregate over intervals of arbitrary length, e.g., finding periods of time where values are monotonically increasing (a temporal *trend* in the data). This should at least support monotonic increases and decreases.

This allows the scenario of UC4.

Finally, we note that UC3 also uses recursion as provided by Datalog, not explicitly given as a requirement. It however, would be hard to meet in a system that does not support recursion.

3.1.2 Time Point Aggregation

In this section, we take on the requirements for moving window and span temporal aggregation we have laid out and introduce our core approach based on declarative operators with implicit representation of time. In particular, we focus on moving window and span temporal aggregation and show efficient algorithms extending the application of the standard non-temporal aggregations [SYZ15] to the temporal context. Time axis aggregation (RQ6) is dealt with in the next subsection.

Moving Windows

The first type of aggregation we discuss are moving window aggregations (e.g., covering the last hour; RQ4). As discussed in Section 2.3, this form of aggregation includes per time point t all the facts that have been valid between $t - w$ and t, where w is some arbitrary window size. For this, we build upon DatalogMTL^{FP} and structure our aggregation into two components. First, we use the $\diamondminus$ operator to extend each time point to the window size w and then we apply the non-temporal aggregation operation [SYZ15] for each time point.

Syntax. So far, DatalogMTL does not contain any operator that enables the aggregation of values. Therefore, we start with the definition of the syntax of time point aggregation for DatalogMTL. by extending the DatalogMTL normal form (cf. preliminaries) with an additional time point aggregation rule of the following form:

$$P_0(\mathbf{s}_0, x) \leftarrow x = aggr(P_1(\mathbf{s}_0, \mathbf{s}_1, a)) \tag{3.1}$$

where $aggr$ is the aggregation type (e.g., count, sum), P_1 and P_0 are predicates, $\mathbf{s}_0$ are the group-by terms, $\mathbf{s}_1$ are the contributor terms for sum and count, and a the aggregate term. In the following, we assume for count that a is the constant term 1. Note that P_1 has arity of size $|\mathbf{s}_0| + |\mathbf{s}_1|$, in case $aggr$ is of type *count*, else $|\mathbf{s}_0| + |\mathbf{s}_1| + 1$ and P_0 has arity $|\mathbf{s}_0| + 1$.

Semantics. We define the semantics of the newly introduced rule on top of monotonic aggregation in plain Datalog [SYZ15]. For this, we consider per time point t a set of all valid facts at t and apply the aggregation rule in plain Datalog over this set of facts:

$$\begin{aligned}
&\mathfrak{M}, t \models^{\nu} x = aggr(P(\mathbf{s}_0, \mathbf{s}_1, a)) \quad \text{if} \\
&\Pi = \{P(\mathbf{s}_0, \mathbf{s}_1, a) \rightarrow \mathit{AggrResult}(\mathbf{s}_0, aggr\langle a, \mathbf{s}_1\rangle)\} \text{ and} \\
&S = \{P(\mathbf{s}) \mid \mathfrak{M}, t \models^{\nu} P(\mathbf{s})\} \text{ and } R = \mathit{Eval}(S, \Pi) \text{ and} \\
&\nu(x) \in \{u \mid \mathit{AggrResult}(\mathbf{s}_0, u) \in R\}
\end{aligned}$$

where $\mathbf{s}$ stands for the sequence of terms $(\mathbf{s}_0, \mathbf{s}_1, a)$, *AggrResult* is a predicate storing the aggregation value, and *Eval* returns the sets of facts resulting from the evaluation of Π on S, using the aggregation in plain Datalog.

Example 3.4. Let us show the benefits of using native temporal operators by expressing UC2 in DatalogMTL extended by monotonic aggregation. Rule 1 extends the interval of *Trade* facts to one hour (assuming a time granularity of seconds) and Rule 2 applies the count operation, using the trader account u as the group-by term and a unique identifier *id* as the contributing term, to derive the number of trades for each time point.

$$\mathit{tradeInterval}(\mathrm{U}, \mathrm{Id}) \leftarrow \diamondminus_{[0,3600)} \mathit{trade}(\mathrm{U}, \mathrm{Id}) \tag{1}$$

$$\mathit{tradeCount}(\mathrm{U}, \mathrm{M}) \leftarrow \mathrm{M} = \mathit{mcount}(\mathit{tradeInterval}(\mathrm{U}, \mathrm{Id})) \tag{2}$$

Algorithm 3.1: Calculation of time point aggregation

Input: an aggregation type T, number of group-by terms g, the aggregation predicate A, and a set of facts $\mathcal{D}'$
Output: A set of aggregated facts B

1 $aggrResult := \emptyset$;
2 $cStorage := \emptyset$;
3 **foreach** $\alpha@\sigma$ *in* $\mathcal{D}'$*; matching predicate of* A **do**
4 a, *groupByKey*, *cKey* = getData(α, A, T, g);
5 $aggrVals = aggrResult[groupByKey]$;
6 $cVals = cStorage[groupByKey][cKey]$;
7 **if** $T = mmin$ **then**
8 UpdateList($aggrVals$, a, σ^-, σ^+, $(a,b) => min(a,b)$);
9 **else if** $T = mmax$ **then**
10 UpdateList($aggrVals$, a, σ^-, σ^+, $(a,b) => max(a,b)$);
11 **else**
12 $changes$:= UpdateList($cVals$, a,σ^-,σ^+,$(a,b) => max(a,b)$);
13 **foreach** $\{e, \sigma_2^-, \sigma_2^+\}$ *in changes* **do**
14 UpdateList($aggrVals$, e, σ_2^-, σ_2^+, $(a,b) => a+b$);
15 $cStorage[groupByKey][cKey]$ = mergeAdjacentInterval($cVals$);
16 $aggrResult[groupByKey]$ = mergeAdjacentInterval($aggrVals$);
17 **return** *aggrResult.values*()

This example immediately highlights the visual and usability advantages of using temporal operators compared to Figure 3.1.

Algorithm. Our algorithm for time point aggregation extends the materialization procedure provided in Algorithm 2.1 to efficiently handle rules with aggregations. Whenever, we have an instance of a rule of form 3.1, we apply Algorithm 3.1 to derive a set of facts (for the head predicate) which are added to the dataset. That is, in comparison to existing DatalogMTL rules, the rule is evaluated using the current dataset, and not on a single grounding of the rule. The algorithm takes as input the type T of the aggregation (e.g., min, max, etc.), the aggregation predicate A, and a number of group-by terms g; it returns as output a set of facts with arity $g+1$, where the first g terms are the group-by terms followed by the aggregated value.

Line 1 defines a map, whose key is the group-by clause, and the value is an ordered set (per time-interval) of the current aggregation values, which, by construction of the algorithm, can contain only non-overlapping time intervals. Line 2 defines a similar map, storing the intermediate results of specific contributor terms. Line 3 iterates over all currently derived facts (denoted as $\mathcal{D}'$ to match the materialization procedure in Algorithm 2.1) filtered by the matching predicate name. Lines 4-6 extract the relevant properties of the fact and retrieve the current sets for the provided keys. In particular, the function *getData* maps the first g terms to the *groupByKey*, the last term to a and the

Algorithm 3.2: UpdateList for temporal aggregation

```
1  Function UpdateList(list, value, intStart, intEnd, aggr):
2      changes := emptyList;
3      if list.isEmpty() then
4          list.append({value, intStart,intEnd});
5          changes.append({value, intStart,intEnd});
6          return changes;
7      it := list.iterator();
8      while list.hasNext() do
9          el := list.next();
10         if intStart > el.intEnd then continue;
11         if intEnd < el.intStart then
12             it.prev();
13             break
14         if intStart < el.intStart then
15             it.prepend({value, intStart, prec(el.intStart)});
16             changes.append({value, intStart, prec(el.intStart)});
17             intStart = el.intStart
18         if intStart > el.intStart then
19             it.prepend({el.value, el.intStart, prec(intStart)});
20             el.intStart = intStart;
21         if intEnd < el.intEnd then
22             it.append({el.value, succ(intEnd), el.intEnd});
23             el.intEnd = intEnd;
24         newValue := aggr(el.value, value);
25         changes.append({newValue - el.value, el.intStart, el.intEnd});
26         el.value = newValue;
27         intStart = succ(el.intEnd);
28     if intStart ≤ intEnd then
29         it.append({value, intStart, intEnd});
30         changes.append({value, intStart, intEnd});
31     return changes
```

remaining terms to the *cKey* (contributor Key). Then the algorithm branches depending on the aggregation type. For instance, we continue with Line 7 for monotonic minimum, Line 9 for monotonic maximum, or with Line 11 for monotonic count and sum. In case of min or max (Line 7-10) we can directly update the final result whereas for count and sum, we first calculate the highest value per contributor (Line 11). As such value can only increase over time, we just consider the difference with respect to the previous contributor value for a certain interval to avoid full recomputation of the aggregate value (Lines 13-14). At the end (Lines 15-16), we iterate over the lists and call mergeAdjacentInterval,

which, as the name says, merges non-overlapping adjacent intervals with the same value to reduce the list size and write the resulting intervals back to the storage. Line 17 returns all output facts of the algorithm (that is, it removes the required grouping for the calculation).

Example 3.5. Consider the aggregation of *tradeInterval* according to Rule (2) in Example 3.4 with the following incrementally added *tradeInterval* to the dataset:

$$\begin{aligned} &\mathit{tradeInterval}(\mathrm{u1}, \mathrm{id1})@[1,6) && \mathit{tradeInterval}(\mathrm{u1}, \mathrm{id2})@[7,12) \\ &\mathit{tradeInterval}(\mathrm{u1}, \mathrm{id3})@[3,8) && \mathit{tradeInterval}(\mathrm{u2}, \mathrm{id4})@[9,14) \\ &\mathit{tradeInterval}(\mathrm{u2}, \mathrm{id5})@[5,10) \end{aligned}$$

Algorithm 3.1 computes the following values (*gK* is *groupByKey*, *cK* is *cKey*):

a	*gK*	*cK*	*aggrVals* (L5)	*changes*	*aggrVals* (L16)
1	u1	id1	$\emptyset$	$[\{1,[1,6)\}]$	$[\{1,[1,6)\}]$
1	u1	id2	$[\{1,[1,6)\}]$	$[\{1,[7,12)\}]$	$[\{1,[1,6)\}, \{1,[7,12)\}]$
1	u1	id3	$[\{1,[1,6)\}, \{1,[7,12)\}]$	$[\{1,[3,8)\}]$	$[\{1,[1,3)\}, \{2,[3,6)\}, \{1,[6,7)\}, \{2,[7,8)\}, \{1,[8,12)\}]$
1	u2	id4	$\emptyset$	$[\{1,[9,14)\}]$	
1	u2	id5	$[\{1,[9,14)\}]$	$[\{1,[5,10)\}]$	$[\{1,[5,9)\}, \{2,[9,10)\}, \{1,[10,14)\}]$

The UpdateList function is detailed in Algorithm 3.2. It iterates over a list of intervals and updates it with the new values. In case the interval list is empty, it just adds the interval (Lines 3-6), otherwise it searches for the first interval starting after the interval to be inserted (Line 10). Then, it checks whether there is some interval to be inserted before the current interval (Lines 14-17), modifies the boundaries of the current entry in case the interval starts or ends in this entry (Lines 18-23), and updates the value of the current entry (Lines 24-27). If the end of the list is reached or the inserting interval ends before the next list entry (Lines 10-12), it adds the remaining interval to the list (Lines 28-30).

Note that storing the aggregation result enables an efficient incremental approach in the algorithm, so that in further applications of the derivation rules, only partial "delta-updates" are computed. For this reason, we skip the initialization of the global variables and keep the current values. In addition, in Line 3 we do not check over all facts, but only over the newly derived ones. We use the function symbols *prec* and *succ* to reference the preceding (resp. succeeding) interval points and use them to harmonize the interval endpoints.

Theorem 3.1. *Algorithm 3.1 has worst-case runtime $O(n^2)$ and worst-case memory consumption $O(n)$, where n is the number of facts contributing to the aggregation. The output has maximum size $O(n)$.*

Proof. We first show the space requirements. For this, we consider UpdateList, which is the only part where new data is added, where we show a space requirement of $O(n)$. Let us start with the *list* which in total contains at most $2n$ entries. There are two different options: (1) the *list* is empty, then the fact is added (Line 3-6), which creates at most one entry for a single fact, or (2) the set is not empty, then the number of added facts depend on existing facts in the *list*. It is clear that Line 14-17 (resp. 18-20), only fire at most once, that is when the new interval to be inserted starts (resp. ends) inside an existing interval. The same applies when the interval starts or ends not in an existing interval, where Line 14-17 add the starting interval before the current entry and Line 28-30 append the ending interval after the current entry. This in total creates at most 2 new entries. It remains to show the additional firings of Line 14-17. If n entries exist in the set, there can be a maximum of $n-1$ gaps. It is clear that a gap is only created when the inserted interval fits between two existing intervals, requiring only 1 new entry. So, in worst case, Line 14-17 only fill up the remaining budget of $2n$ entries. Let us now consider *changes*. In worst case, when the new interval spans over the complete *list*, then each entry in the *list* requires one *changes* entry, that is in total at most 2n entries. In total, this yields $O(2n+2n) = O(4n) = O(n)$ as the space bound.

For runtime, we can build upon the space requirements. It is clearly visible that the runtime for *mmin* and *mmax* equals to the first part of *msum* and *mcount*, i.e., the update operation of line 11 and 9 are equal for *mmax* or similar for *mmin*. Hence, we only have to consider the else part inside the outer foreach loop. For each fact ($O(n)$), we update the contributor set (Line 13) and then apply updates to the aggregation value (Line 15-16). We start by showing the runtime of the update for the contributor set. It is clear that UpdateList iterates once over the list, which runs in $O(2n)$ in worst case. We now consider the update process and show that we will not exceed the runtime of the updating process. We have shown that the space bound of *aggrValues* as well as *changes* is $O(2n)$. Since those sets are ordered, we can scan both sets together within an one-pass iteration by checking the boundaries of the following intervals. Hence, we do not exceed the runtime of $O(2n)$. The same applies to the merging operations, which iterates over the list once. This concludes the argument that the algorithm takes $O(2n) = O(n)$ time in worst case for a single added value. Now we consider that n facts contribute to the aggregation, hence the algorithm is executed n times and thus runs in $O(n^2)$. The final flattening step loops over the *aggrResult* of size $O(2n)$ and maps the result to the appropriate format of the output. Hence, the output size is also at most $O(n)$. □

Theorem 3.2. *Algorithm 3.1 is sound and complete.*

Proof. Let us start with the completeness of the algorithm. We have to show that for each aggregation predicate in the dataset and for each interval of the aggregation predicate a value is derived in the result. The only place where we add values to the result is inside of UpdateList, which is executed with each fact and interval in the dataset. On a high-level this function iterates over the list and inserts values into the list whenever the

current fact contains a range in the interval which is currently not covered in the list. There exist multiple cases of how two intervals can overlap:

- *The list is empty.* The full interval is added and returned (Lines 3-6)
- *The interval starts but not ends before the next list entry.* The interval is inserted up to the start of the next list entry (Lines 14-17) and the start value is then updated to the end of the next list entry (Line 27).
- *The interval starts inside a list entry.* The interval is split (Lines 18-20) and the start value is then updated to the end of the next list entry (Line 27).
- *The (remaining) interval starts after the last list entry.* The interval is inserted (Lines 28-30).
- *The (remaining) interval ends between list entries.* The interval is inserted (Lines 28-30).
- *The interval ends inside a list entry.* The interval is split (Lines 21-23) and the start value is then updated to the end of the next list entry (Line 27).

At the end of this loop, by the above cases, there exists a value for each time point in the interval, thus the algorithm is complete.

Let us now consider the soundness of the algorithm. We have to show that each derived interval contains the correct value. This is for *mmin* the lowest possible value and for *mmax*, *msum* and *mcount* the highest possible value at each time point. Following the argument for completeness, the only place where we change the values of one interval is inside of UpdateList, which is executed with each fact in the dataset. In detail, this update is given by the chosen *aggr*-function in Line 24 of UpdateList that calculates the correct value for each interval. This function is triggered for each already existing entry that may change (due to the splitting in Lines 18-20 and 21-23) or missing entry of the list, thus the algorithm is sound.

□

Spanning Intervals

The next type of aggregation we discuss is span temporal aggregation (i.e., aggregating values per period, e.g., months that vary between 28 and 31 days). While for moving windows, we could use an existing temporal operator to extend the interval to the desired window size, for span temporal aggregation no such operator exists in DatalogMTL. Therefore, we introduce an additional granularity operator that extends intervals to their complete unit of interest, e.g., an interval from the 15th of March to 17th of April to the 1st of March to the 30th of April. Such kinds of operators have been successfully applied in the context of temporal databases (cf., Bettini et al. [BJW00]).

Syntax. We extend the DatalogMTL normal form to formally define the new granularity operator with an additional rule of the following form:

$$P_0(\mathbf{s}_0) \leftarrow \bigtriangleup_{unit} P_1(\mathbf{s}_1) \tag{3.2}$$

where $\bigtriangleup$ is the new granularity operator that extends the interval to the full period of an interest and *unit* is the time unit of interest, e.g., day, month, or year.

Semantics. The semantics of the operator $\bigtriangleup$ is defined as follows:

$$\mathfrak{M}, t \models^{\nu} \bigtriangleup_{unit} A \quad \text{if } \mathfrak{M}, s \models^{\nu} A \text{ for some } s \text{ with } conv(t, unit) = conv(s, unit)$$

where *conv* converts the time point to the provided time unit. For example, the date 31.12.2020 with unit year, would be converted to 2020.

Example 3.6. Let us demonstrate the span temporal aggregation by expressing UC1. Rule 1 extends the interval of the sales predicate to its corresponding year and Rule 2 applies the sum operation, using the Id as contributing term and Price as aggregation term, to derive the revenue of the year.

$$yearSale(\text{Id}, \text{Price}) \leftarrow \bigtriangleup_{year} sale(\text{Id}, \text{Price}) \tag{1}$$

$$revenue(\text{M}) \leftarrow \text{M} = msum(yearSale(\text{Id}, \text{Price})) \tag{2}$$

Looking Ahead. With the introduced granularity operation, we are able to extend intervals to periods of interest and combined with the DatalogMTL operators we are able to offset and extend the periods of interest. Yet, there exist typical operations that require more advanced reasoning techniques. For example, in the financial domain we wish to compare the revenue of a year on a weekday basis. While it is possible to some extend to handle this with the available operations (e.g., grouping facts per weekday with a join and then extending them to a year as visualized in Example 3.7), we suggest for such requirements the introduction of unwrapping operations that map the timestamp to an appropriate representation in plain Datalog, e.g., by introducing rules of the following form:

$$saleEscaped(\text{Id}, \text{Price}, weekday(\text{t})) \leftarrow Sale(\text{Id}, \text{Price})@t$$

An implementation of the unwrapping operation is considered in Chapter 5.

Example 3.7. In DatalogMTL the given example of computing the revenue basis of

a year on a weekday basis could be formulated in the following way:

$$
\begin{aligned}
&\mathit{weekday}(\text{monday})@[20220103, 20220104)\\
&\mathit{weekday}(\text{tuesday})@[20220104, 20220105)\\
&\dots\\
\mathit{weekday}(\mathrm{W}) &\leftarrow \diamondminus_{[7d,7d]}\, \mathit{weekday}(\mathrm{W})\\
\mathit{saleGroup}(\mathrm{W}, \mathrm{Id}, \mathrm{Price}) &\leftarrow \mathit{sale}(\mathrm{Id}, \mathrm{Price}), \mathit{weekday}(\mathrm{W})\\
\mathit{yearSaleGroup}(\mathrm{W}, \mathrm{Id}, \mathrm{Price}) &\leftarrow \bigtriangleup_{\mathit{year}} \mathit{saleGroup}(\mathrm{W}, \mathrm{Id}, \mathrm{Price})\\
\mathit{revenue}(\mathrm{W}, \mathrm{M}) &\leftarrow \mathrm{M} = \mathit{msum}(\mathit{yearSaleGroup}(\mathrm{W}, \mathrm{Id}, \mathrm{Price}))
\end{aligned}
$$

3.1.3 Time Axis Aggregation

So far, we have focused on aggregating values per time point. In this section, we now move to aggregations along the time axis. As introduced in UC4 and RQ6, this means we need to consider adjacent time points and summarize their changes over time. In this work we focus on one concrete type of time axis aggregation that was raised as important requirement by industrial stakeholders from the financial domain, namely the detection of whether a trend is monotonically increasing/decreasing over time. Let us call it *minc* resp. *mdec*. This is effective in many domains, e.g., that of change of control we have introduced, but also for example population counts. We first start with an example and consider the desired behavior.

In the following, we assume that the domain of this type of time axis aggregation is that of disjoint intervals, and so, for a single group-by key, no ambiguity arises w.r.t. the aggregation term. This makes time axis aggregation fully orthogonal to time point aggregation, where, instead, intervals are combined. Also, time point aggregation can be effectively used to disambiguate values per time interval (e.g., by considering their maximum/minimum resp. the summation) before proceeding with time axis aggregation.

Example 3.8. Assume that we want to detect changes of control as described in UC4. Then, we are interested in finding all intervals in which the number of shares has been monotonically increasing as well as the minimum and maximum values in those intervals. Assume now that the atom $\mathit{shares}(\mathrm{P}, \mathrm{C}, \mathrm{S})$ represents the number of shares S that an investor P owns of a company C. Then a possible operator has to take as argument the number of shares, groups them by investor and company and returns as output the lower bound (i.e., the leftmost value) and the upper bound (i.e., the rightmost value) per monotonically increasing interval. This is exactly shown in Rule 1, which applies the monotonic increasing operator (*minc*) to derive the lower and upper point of the monotonic increasing interval.

$$\mathit{sInc}(\mathrm{P}, \mathrm{C}, \mathrm{L}, \mathrm{U}) \leftarrow \langle \mathrm{L}, \mathrm{U} \rangle = \mathit{minc}(\mathit{shares}(\mathrm{P}, \mathrm{C}, \mathrm{s})) \tag{1}$$

As for time point aggregation, in the following we (i) provide a syntax for time axis aggregation, (ii) specify the semantics, and (iii) suggest an algorithm for the operations.

Syntax. For time axis aggregation, we build upon the syntax of time point aggregation (Equation (3.1)) and generalize it to a generic form supporting multiple output values. For this, we extend the DatalogMTL normal form with an aggregation rule of the following form (actually already used in Example 3.8):

$$P_0(\mathbf{s}_0, \mathbf{x}) \leftarrow \langle \mathbf{x} \rangle = aggr(P_1(\mathbf{s}_0, \mathbf{s}_1, a)) \tag{3.3}$$

Functor *aggr* is the name of the time axis aggregation, in our case *minc* or *mdec*, and P_1 and P_0 are predicates. Like in time point aggregation, $\mathbf{s}_0$ defines the group-by clause, $\mathbf{s}_1$ the contribution terms (not used for detecting monotonic trends but kept for uniformity reasons), and a is the aggregation term of P_1. Over time, there can be multiple values of interest, for example the start and end value of a monotonically increasing interval. Hence, the function returns a vector $\mathbf{x} = x_1, \ldots, x_n$ of aggregation values instead of a single value. For *minc* and *mdec* we use $\mathbf{x} = \mathrm{L}, \mathrm{U}$ to denote the lower and upper bound of the monotonic interval.

Semantics. With the requirement of using disjoint intervals, the semantics for detecting monotonic trends can be easily formulated by moving from time points to time intervals, i.e., $\mathfrak{M}, \sigma \models^{\nu} \phi$ if $\mathfrak{M}, t \models^{\nu} \phi$ for all $t \in \sigma$ as we see in the following semantic definition:

$$\begin{aligned}
&\mathfrak{M}, \sigma \models^{\nu} \langle l, u \rangle = minc(P_1(\mathbf{s}_0, a)) && \text{if } \mathfrak{M}, \sigma \models^{\nu} M(P_1, \mathbf{s}_0, l, u) \\
&\mathfrak{M}, \sigma \models^{\nu} M(P_1, \mathbf{s}_0, a, a) && \text{if } \mathfrak{M}, \sigma \models^{\nu} P_1(\mathbf{s}_0, a) \\
&\mathfrak{M}, \sigma \models^{\nu} M(P_1, \mathbf{s}_0, l, u) && \text{if } \mathfrak{M}, \sigma_1 \models^{\nu} M(P_1, \mathbf{s}_0, l, u_1) \text{ and} \\
& && \quad \mathfrak{M}, \sigma_2 \models^{\nu} M(P_1, \mathbf{s}_0, l_2, u) \text{ and} \\
& && \quad u_1 \leq l_2 \text{ and } \sigma_1^+ \prec \sigma_2^- \text{ and } \sigma = \sigma_1 \cup \sigma_2
\end{aligned}$$

where $\prec$ is the predecessor relation (i.e., the intervals are adjacent), and M a fresh predicate for deriving *minc*. In short, the second definition states that a constant value in an interval is both lower and upper bound, and the third one merges two intervals if their lower and upper bounds match. The semantics for *mdec* is analogous, with $u_1 \leq l_2$ changed to $u_1 \geq l_2$.

Algorithm. Like for time point aggregation, we provide an efficient algorithm for *minc* and *mdec* which uses the benefits of native temporal operators. Algorithm 3.3 takes as input the aggregation predicate A, a number of group-by indices g and the currently derived set of facts $\mathcal{D}'$ and returns as output a set of facts B. Line 1 defines an empty map for the result, where the keys are the group-by terms, and the values are B-trees with the intervals as key of the entries. Then for each fact, we add the fact to the appropriate group-by clause (Lines 3-5). We then merge the inserted fact (Line 7-12) with their adjacent intervals, if they exist, so that we derive the largest possible, monotonically increasing interval. For deriving monotonically decreasing intervals, the comparison operator in Lines 7 and 11 has to be changed from $\leq$ to $\geq$. Line 13 returns all output

Algorithm 3.3: Calculation of monotonic monotonically increasing intervals

Input: number of group-by terms g in α, and the aggregation predicate A, a set of facts $\mathcal{D}'$
Output: A set of aggregated facts B

1 $B := \emptyset$;
2 **foreach** $\alpha@\sigma$ *in* $\mathcal{D}'$*; matching predicate of* A **do**
3 a, *groupByKey* := getData(α,A,g);
4 *aggrGroup* := X[*groupByKey*];
5 *nNode* := *aggrGroup*.insert($(a, a)\#\sigma$);
6 *lNode* := *nNode*.previous;
7 **if** $\sigma^- = lNode.\sigma^+ \wedge lNode.max \leq a$ **then**
8 Remove *lNode*, *nNode* from *aggrGroup*;
9 *nNode* := Insert $(lNode.min, nNode.max)\#\langle lNode.\sigma^-, \sigma^+\rangle$ to *aggrGroup*;
10 *rNode* := *nNode*.next;
11 **if** $\sigma^+ = rNode.\sigma^- \wedge a \leq rNode.min$ **then**
12 Apply merging for *rNode* similar to *lNode*;
13 **return** *X.values()*

facts of the algorithm (that is, it removes the required grouping for the calculation). Note that, similarly to Algorithm 3.1, delta-updates can be applied by skipping Line 1 of the algorithm.

Example 3.9. Consider the computation of monotonic trends according to Example 3.8 with the following dataset:

$$\mathit{shares}(\text{p1}, \text{c1}, 5)@[4, 7) \qquad \mathit{shares}(\text{p1}, \text{c1}, 8)@[7, 9) \qquad \mathit{shares}(\text{p1}, \text{c1}, 3)@[3, 4)$$

Algorithm 3.3 computes the following values:

a	*groupByKey*	*aggrGroup* (L5)	*aggrGroup* (L12)
5	(p1,c1)	$[(5, 5)\#[4, 7)]$	$[(5, 5)\#[4, 7)]$
8	(p1,c1)	$[(5, 5)\#[4, 7),\ (8, 8)\#[7, 9)]$	$[(5, 8)\#[4, 9)]$
3	(p1,c1)	$[(3, 3)\#[3, 4),(5, 8)\#[4, 9)]$	$[(3, 8)\#[3, 9)]$

Theorem 3.3. *Algorithm 3.3 has runtime* $O(n\log(n))$ *in the worst case and worst-case memory consumption* $O(n)$*, where* n *is the number of facts contributing to the aggregation. The output has maximum size* $O(n)$*.*

Proof. First, we show the space requirement of $O(n)$. Each fact can be added to the tree at most once (Line 5). This state is reached if the facts are strictly monotonically decreasing when monotonically increasing is asked. The other lines remove more facts than those added, hence the size of the list only reduces. This ends the argument of space

requirement of at most $O(n)$. This also shows the output size of $O(n)$ of the algorithm. Now, we show the runtime complexity of $O(n\log(n))$. Inserting and removing facts in a B-tree has a complexity of $O(\log(n))$. We then consider that we have to execute this for each fact contributing to the aggregation, i.e., $O(n)$ times. This yields a runtime complexity of $O(n\log(n))$. □

Theorem 3.4. *Algorithm 3.3 is sound and complete.*

Proof. Let us start with completeness, which follows immediately. Since all values (with their intervals) are inserted in the tree, the algorithm does not miss any case. We now show the soundness of the algorithm. We show this by induction. In the base case, we add the first interval to a group-by key, which is simply the value of the fact. We now assume that we have maximal monotonic increasing intervals and that we add another fact. Since the considered facts must have a disjunct interval by definition, there are three cases, we have to distinguish.

- The added interval is not adjacent to any neighbor. Then we are done, and we have the maximum monotonic increasing interval.
- The interval is adjacent to the left neighbor. If the two monotonic intervals are increasing, then by Line 7-9, we merge the two neighbors creating a new bigger maximal interval over the currently maximal interval of the left neighbor and the added interval and update the bounds.
- The interval is adjacent to the right neighbor. We have two cases, either the inserted interval is not adjacent with the left neighbor, or the interval is also adjacent with the left neighbor. In both cases, we assume that the current interval is already the maximal interval regarding the left boundary (i.e., it has already been merged with the left neighbor in case the intervals are adjacent and monotonic (see Line 9)). By Line 11, we merge the current maximal interval with the right interval in case they are monotonic, creating the biggest monotonic interval.

Hence, we derive always the maximum interval, and the algorithm is sound and complete. □

3.2 DatalogMTL with existential quantification

Existential rules allow the invention of new values in the reasoning process. This is especially interesting in the domain of knowledge graph reasoning as it allows to enrich incomplete data with domain knowledge and is heavily applied among other uses in the Vadalog system for non-temporal reasoning [BSG18].

The usage of existential rules for temporal reasoning has been studied in the context of temporal description logics [AF05] and atemporal languages with linear-order operators [ABBV18], which can simulate some form of temporal reasoning. However, there is

a lack of comprehensive research on highly expressive temporal extensions of Datalog as decidability was obtained only in very restrictive cases, for example when the temporal domain is bounded [UKE22].

In this section, we introduce DatalogMTL$^{\exists}$, an extension of DatalogMTL with existential rules. In Section 3.2.1 we define the syntax and two different semantics, a natural one, where existential rules are evaluated at each time point, and a uniform one, where an existential variable of a rule gets assigned the same constant for each time point. Then, we provide the main complexity results for the given fragments in Section 3.2.2. Finally, in Section 3.2.3 we provide an extension of the materialization algorithm of Section 2.2 to existential quantification for the only decidable fragment under OWA.

Note that as mentioned earlier, we mainly contributed to the applied results of the studying of existential quantification, thus we refer an interested reader to the paper for the theoretical details [LNSW23] as well as to an extended abstract of our colleagues [LW22].

3.2.1 Syntax and Semantics of DatalogMTL$^{\exists}$

Similar to existential rules in Datalog, we extent DatalogMTL with (temporal) existential rules to obtain DatalogMTL$^{\exists}$.

Formally, an existential rule is given as

$$\exists \mathbf{X}\ M' \leftarrow M_1 \wedge \cdots \wedge M_n, \qquad \text{for } n \geq 0,$$

where $\mathbf{X}$ is a tuple of variables mentioned in M' but not in the rule body. Similar to Datalog, we usually omit $\exists \mathbf{X}$ when writing a rule. The other definitions (e.g., safe rules, program, grounding, and so on) apply analogously to DatalogMTL, usually considering only free variables (i.e., variables that are not existential quantified). This means for example that a rule is safe if all its free variables are mentioned in the body or that the grounding assigns constants to free variables.

Regarding the semantics, we distinguish between two types, a 'natural' semantics (N), which is based on the standard reading of existential quantification (i.e., applying existential quantification in the usual way, e.g., as in Datalog$^{\exists}$, also for DatalogMTL) and a 'uniform' semantics (U) that avoids some intuitive situations for temporal reasoning: A rule in natural semantics can invent, for an interval ϱ in which the body holds, distinct constants for each $t \in \varrho$. This is shown in Example 3.10.

Example 3.10. Consider the following program Π_{ex}, modelling that if a person X is CEO, that they have to work for a company C

$$\mathit{work}(\mathrm{X}, \mathrm{C}) \leftarrow \mathit{person}(\mathrm{X}, \mathrm{ceo})$$

and a given dataset $\mathcal{D}_{\mathsf{ex}}$ consisting of a single fact $\mathit{person}(\mathrm{p1}, \mathrm{ceo})@[2008, 2012]$.

The natural semantics allows models where a person works for different companies

for each time instant in the interval $[2008, 2012]$. Under uniform semantics there is a single model where a person works for a company $c1$ between 2008 and 2012.

In detail, the definitions are:

Definition 3.2.1. Under the *natural semantics*, an interpretation $\mathfrak{I}$ satisfies an existential rule r, written as $\mathfrak{I} \models_N r$, if for each ground rule $r' \in \mathsf{ground}(\{r\})$ of the form $\exists \mathbf{X}\ M' \leftarrow M_1 \wedge \cdots \wedge M_n$ and every time point t in which $\mathfrak{I}$ satisfies all body atoms $M_1, \ldots, M_n$, there exists an assignment ν of constants in Dom to variables of M' such that $\mathfrak{I}, t \models \nu(M')$. Interpretation $\mathfrak{I}$ satisfies a program Π under the natural semantics, written as $\mathfrak{I} \models_N \Pi$, if and only if $\mathfrak{I} \models_N r$, for each $r \in \Pi$.

Definition 3.2.2. Under the *uniform semantics*, an interpretation $\mathfrak{I}$ satisfies an existential rule r, written as $\mathfrak{I} \models_U r$, if for every atom M' there exists an assignment $\nu_{r,M'}$ of constants to variables of M' such that for each ground rule $r' \in \mathsf{ground}(\{r\})$ of the form $\exists \mathbf{X}\ M' \leftarrow M_1 \wedge \cdots \wedge M_n$ and every time point t in which $\mathfrak{I}$ satisfies all body atoms $M_1, \ldots, M_n$, we have $\mathfrak{I}, t \models \nu_{r,M'}(M')$. Interpretation $\mathfrak{I}$ satisfies a program Π under the uniform semantics, written as $\mathfrak{I} \models_U \Pi$, if $\mathfrak{I} \models_U r$ for each $r \in \Pi$.

The semantics relate in such a way that if a fact is entailed under natural semantics by some program and dataset, then it will also be entailed under uniform semantics, as Proposition 3.5 shows:

Proposition 3.5. *Let* Π *be a* DatalogMTL$^{\exists}$ *program,* $\mathcal{D}$ *a dataset, and* $P(\mathbf{s})@\varrho$ *a fact. If* $(\Pi, \mathcal{D}) \models_N P(\mathbf{s})@\varrho$*, then* $(\Pi, \mathcal{D}) \models_U P(\mathbf{s})@\varrho$*, but the opposite implication does not hold. This property holds under both OWA and CWA.*

This proposition can be exemplified as follows:

Example 3.11. Consider the reasoning of Example 3.10 again. As the natural semantics allows multiple models, it does not entail the fact $\mathit{work}(\mathrm{p1}, \mathrm{c1})@[2008, 2012]$, i.e., $\Pi_{\mathsf{ex}}, \mathcal{D}_{\mathsf{ex}} \not\models_N \mathit{work}(\mathrm{p1}, \mathrm{c1})@[2008, 2012]$. However, under uniform semantics, which enforces the same existential quantification for all time instants, there is a single model, entailing the fact $\Pi_{\mathsf{ex}}, \mathcal{D}_{\mathsf{ex}} \models_U \mathit{work}(\mathrm{p1}, \mathrm{c1})@[2008, 2012]$.

Furthermore, if we forbid the use of temporal operators, DatalogMTL$^{\exists}$ under either semantics behaves exactly like Datalog$^{\exists}$, as we state formally next.

Proposition 3.6. *Let* $P(\mathbf{s})@0$ *be a fact,* Π *a* DatalogMTL$^{\exists}$ *program not mentioning any temporal operators,* $\mathcal{D}$ *a set of facts of the form* $P(\mathbf{s})'@0$*, and* $\mathcal{D}' = \{P(\mathbf{s})' \mid P(\mathbf{s})'@0 \in \mathcal{D}\}$*. Then the following are equivalent: (i)* $\Pi, \mathcal{D} \models_N P(\mathbf{s})@0$ *under OWA, (ii)* $\Pi, \mathcal{D} \models_U P(\mathbf{s})@0$ *under OWA, and (iii)* Π *and* $\mathcal{D}'$ *entail* $P(\mathbf{s})$ *in* Datalog$^{\exists}$*.*

3.2.2 Complexity results

In this section, we summarize the complexity results. Clearly, undecidability of N- and U-consistency in full DatalogMTL$^{\exists}$ follows immediately from the well-known undecidability

of Datalog$^{\exists}$. Thus, our results consider the guarded and weakly-acyclic fragments, two useful practical fragments requested by stakeholders, which are both decidable in Datalog$^{\exists}$. Note that checking $\mathfrak{S}$-consistency and $\mathfrak{S}$-entailment reduce in logarithmic space to the complement of each other, where $\mathfrak{S} \in \{U, N\}$.

We observe that N-consistency is undecidable in each of these fragments, even under the restrictive case of CWA where existentially quantified variables are bound only to constants in ADom.

Theorem 3.7. *Under both OWA and CWA, checking N-consistency for guarded as well as for weakly acyclic* DatalogMTL$^{\exists}$ *programs is undecidable.*

Under uniform semantics, we observe a similar result for guarded programs under OWA, which implies undecidability of many other well-known fragments, such as weakly-guarded or frontier-guarded programs. The decidability of less expressive formalisms are at the moment unknown and up to further research.

Theorem 3.8. *Checking U-consistency is undecidable for guarded* DatalogMTL$^{\exists}$ *programs under OWA.*

Note that the undecidability under CWA can be shown by a reduction to the undecidability of consistency checking of DatalogMTL with $\oplus_{(0,1)}$ in the head [BKK+17a]. For the case of OWA, one can create a program which provides access to infinitely many constants (one constant per (integer) time point) and thus allows to simulate a Turing machine. As only integers are used in the proof, we further observe that all the undecidability proofs apply also to the case when the timeline consists of integer time points only. In contrast, all uniform DatalogMTL$^{\exists}$ programs become decidable for CWA as well as for weakly acyclic programs under OWA.

Theorem 3.9. *Under OWA, checking U-consistency is 2-*EXPSPACE*-complete for weakly acyclic* DatalogMTL$^{\exists}$ *programs.*

Theorem 3.10. *Under CWA, checking U-consistency is* EXPSPACE*-complete for arbitrary* DatalogMTL$^{\exists}$ *programs.*

Note that the upper bounds for the decidability can be shown by a reduction to a ground DatalogMTL program by carefully constructing a set of constants (with nulls) that is used to non-deterministically guess assignments for interpreting existential rules. The EXPSPACE-hardness is inherited from DatalogMTL and the 2-EXPSPACE-hardness by constructing a simulation of a Turing machine with doubly-exponentially many tape cells.

A summary of the results is provided in Table 3.1.

Table 3.1: Summary of complexity results for DatalogMTL$^{\exists}$

		DatalogMTL$^{\exists}$	G. DatalogMTL$^{\exists}$	W.-A. DatalogMTL$^{\exists}$
Natural Semantics	OWA	undecidable		
	CWA			
Uniform Semantics	OWA			2-EXPSPACE-complete
	CWA	EXPSPACE-complete		

3.2.3 Implementation

In this section, we provide the first approach for reasoning in weakly acyclic DatalogMTL$^{\exists}$ programs under uniform semantics, which is based on materialization (see Algorithm 2.1). It is worth noting that naïve materialization of DatalogMTL (and so DatalogMTL$^{\exists}$) programs, as applied in this algorithm, is not guaranteed to terminate [WTCKCG21]. However, such a procedure is sufficient for reasoning in a wide range of practical problems and many other techniques that involve materialization as a sub-procedure.

To support DatalogMTL$^{\exists}$ programs we propose the adaption of a well-known technique for weakly-acyclic Datalog$^{\exists}$ programs [Mar09, BKM^{+}17] called *Skolemization*, in which existential variables z in rules are replaced by Skolem terms $f_z(\mathbf{X})$ that depend only on the frontier variables $\mathbf{X}$ (i.e., those variables that are shared between the head and the body of a rule) and where f is a unique function symbol per existential variable in a rule head.

Example 3.12. Consider the rule of Example 3.10. The existential variable C in the rule is replaced by $r_{1C}(\mathrm{X})$, where r_1 is a unique id of the rule, resulting in the rule:

$$work(\mathrm{X}, r_{1C}(\mathrm{X})) \leftarrow person(\mathrm{X}, \mathrm{ceo})$$

The full materialization process is provided in Algorithm 3.4, which adapts the materialization approach for DatalogMTL to support interpretation of existential rules under the uniform semantics. The procedure takes a DatalogMTL$^{\exists}$ program Π and a dataset $\mathcal{D}$ as input, and returns their materialization , if it terminates, or detects that Π and $\mathcal{D}$ are inconsistent. To simplify the presentation, we assume without loss of generality that Π is in normal form (i.e., has no temporal operators in rule heads). First, the existential rules of Π are rewritten according to the procedure described before by introducing Skolem terms for all existentially quantified positions. Then the procedure performs in a loop, consequent materialization steps (Lines 4–17) until inconsistency is detected (Line 13) or the full materialization is obtained, that is, there are no more facts to be derived (Line 17). The loop (Lines 5–16) first derives the new facts per grounded rule (Line 6), thereby replacing the Skolem terms with constants according to a maintained set of assignments. The set of assignments is built up successively, such that newly encountered Skolem terms are mapped to fresh constants, while previously seen terms map to the same constant as before (Lines 7–10), thus enforcing uniformity. Note that since the loop in Line 5 is over ground rules, the arguments of the Skolem terms are also constants. Then

Algorithm 3.4: Materialization under the uniform semantics

Input: A DatalogMTL$^{\exists}$ program Π and a dataset $\mathcal{D}$
Output: A dataset representing materialization of Π and $\mathcal{D}$, or 'inconst' if Π and $\mathcal{D}$ are inconsistent

1 $\Pi_S \leftarrow$ Skolemisation of Π;
2 $\mathcal{D}' \leftarrow \mathcal{D}$;
3 *assignments* $\leftarrow$ empty set of assignments;
4 **repeat**
5 **for** *each* $r \in \mathsf{ground}(\Pi_S, \mathcal{D}')$ *and a (maximal) interval* ϱ *at which* $\mathcal{D}'$ *entails the body of* r **do**
6 $H \leftarrow$ head of r;
7 **for** *each Skolem term* $f_z(\mathbf{c})$ *in* H **do**
8 **if** *no assignment for* $f_z(\mathbf{c})$ *in assignments* **then**
9 $d \leftarrow$ a fresh constant ;
10 Add $f_z(\mathbf{c}) \mapsto d$ to *assignments*;
11 Replace Skolem terms in H according to *assignments*;
12 **if** H *mentions* $\bot$ **then**
13 **return** inconst;
14 $\mathcal{D}' \leftarrow \mathcal{D}' \cup \{H\}$;
15 Coalesce facts in $\mathcal{D}'$;
16 **until** *no change to* $\mathcal{D}'$;
17 **return** $\mathcal{D}'$;

(Lines 12–13), the derived head is checked for any inconsistency. Finally, the derived facts are coalesced (Line 15), that is, facts with overlapping or adjacent intervals ϱ_1, ϱ_2 are recursively replaced with a fact over an interval $\varrho_1 \cup \varrho_2$.

Theorem 3.11. *Consider Algorithm 3.4 running on inputs* Π *and* $\mathcal{D}$*. If the output is* inconst*, then* Π *and* $\mathcal{D}$ *are inconsistent. Otherwise they are consistent, and for every fact* $M@\varrho$ *mentioning only constants from* Π *and* $\mathcal{D}$*, we have* $\mathcal{D}' \models M@\varrho$ *if and only if* $\Pi, \mathcal{D} \models_U M@\varrho$*, where* $\mathcal{D}'$ *is the output dataset.*

Proof sketch. We argue soundness by showing inductively that for every satisfying interpretation $\mathfrak{I}$ of Π and $\mathcal{D}$ under uniform semantics, there is a certain type of temporal homomorphism from $\mathcal{D}'$ to $\mathfrak{I}$, akin to the role of universal models in Datalog$^{\exists}$ (cf. [DNR08]). To observe correctness in the case where the output is $\mathcal{D}'$, it suffices to verify that the returned $\mathcal{D}'$ is always a satisfying interpretation of Π and $\mathcal{D}$ under the uniform semantics. □

It is worth observing that Algorithm 3.4 provides a sound reasoning approach for the uniform semantics, but not for the natural semantics. Indeed, our Skolemisation approach

introduces the same Skolem terms for all time points which satisfy the body of a ground rule, which is in the spirit of Definition 3.2.2, but not of Definition 3.2.1.

3.3 Summary

In this chapter, we introduced two key capabilities for knowledge graph reasoning missing in DatalogMTL. In the first part we extended DatalogMTL with various kinds of aggregations. We focused on adding support for instantaneous, moving window and span temporal aggregation as well as on time-axis aggregation, in particular on detecting monotonic increasing and decreasing intervals. To this end, we provided algorithms that efficiently integrate into the materialization process. In the second part we extended DatalogMTL with existential quantification, a necessary extension for generating new entities in knowledge graphs. We observed two different kinds of semantics and provided an adapted materialization algorithm for weakly-acyclic programs under the uniform semantics, the only fragment that has been shown to be decidable under OWA. Together, these results are the foundations for enabling expressive reasoning capabilities for temporal knowledge graph systems.

CHAPTER 4

Evaluation Tooling - Fundamental Benchmark Generator

Having laid the foundation for temporal knowledge graph reasoning in the previous chapter, the importance to have efficient systems that interpret and execute DatalogMTL sparks the consequential and dire need for benchmarks. This is both to measure the performance of reasoning in such systems and to stimulate and evaluate the theoretical underpinnings of the language at hand.

While there exist benchmarks for various kinds of use cases involving temporal data and reasoning as introduced in Section 2.5, some of these benchmarks are not publicly available, and nearly all of them share the common problem of neglecting recursion, a key requirement of time-based reasoning, as we have seen and will explore further. In particular, nearly none of the existing benchmarks tests full recursion. Also, their coverage of aggregation is highly insufficient, as the performance of aggregation can be measured only independently of time and in non-recursive settings.

Hence, there is a widespread wish and need for a well-established benchmark in the area around DatalogMTL in order to ensure fair system competition. While there has been a lot of work around DatalogMTL so far, benchmarks have been recognized as the one missing piece to put well-established theory into practice.

In this chapter, we introduce iTemporal, a benchmark generator targeted for generating DatalogMTL programs, including the discussed extensions in the previous chapter. This generator is not just a set of benchmark instances, but a flexible benchmark generator of both rules and data that can be configured for various scenarios to explore how the systems capture the theoretical underpinnings of DatalogMTL.

This chapter is based on our ICDE paper [BNS22], were we first introduced the generator, and an extension for existential quantification established during the work on

DatalogMTL$^{\exists}$. The remainder of this chapter is organized as follows: We discuss along the lines of a running example the key capabilities of a benchmark generator in Section 4.1. Then, in Section 4.2 we introduce the core of our benchmark generator, followed by the aggregation module in Section 4.3 and the existential quantification module in Section 4.4. Finally, we summarize the chapter in Section 4.5.

4.1 Requirements

In this section, we analyze the requirements for generating benchmarks of DatalogMTL. We first consider Example 4.1, a running example, we will use throughout this chapter. This example naturally uses the components mentioned so far – (1) recursion, (2) temporal operators, (3) aggregation, and (4) existential quantification.

Example 4.1. The following rules feed a resource management dashboard monitoring total expenses after each transaction.

$$\mathit{expense}(\mathrm{Id}, \mathrm{P}) \leftarrow \mathit{salary}(\mathrm{Id}, \mathrm{P}) \tag{1}$$
$$\mathit{expense}(\mathrm{Id}, \mathrm{F}) \leftarrow \mathit{noResp}(\mathrm{Id}) \mathcal{S}_{[24,24]}\, \mathit{claim}(\mathrm{Id}, \mathrm{F}) \tag{2}$$
$$\mathit{expenses}(\mathrm{T}) \leftarrow \mathit{expense}(\mathrm{Id}, \mathrm{P}), \mathrm{T} = \mathit{msum}(\mathrm{P}) \tag{3}$$
$$\mathit{expense}(-1, \mathrm{T}) \leftarrow \diamondminus_{(0,1]}\, \mathit{expenses}(\mathrm{T}) \tag{4}$$
$$\mathit{report}(\mathrm{Id}, \mathrm{T}) \leftarrow \mathit{expenses}(\mathrm{T}) \tag{5}$$

The first two rules model possible types of expenditures for a company (indeed, many more exist in practice). In particular, Rule 1 accounts for expenditures that correspond to salaries, each characterized by an Id and an amount P to be paid. Rule 2 deals with claims: if a *Claim Id* of value F receives no response within 24 hours, as denoted by the since ($\mathcal{S}$) operator, it will generate a new expenditure. The goal of Rule 3 is to calculate the sum (T) over all *Expense* items per timepoint t, by summing their amount P. Rule 4 interacts with Rule 3, propagating partial sums through time by jointly using the $\diamondminus_{(0,1]}$ operator and recursion. In particular, whenever there are expenses in the $[t-1, t)$ interval, their total T is carried as a partial contributor for the evaluation at t of Rule 3. Rule 5 introduces an existential quantified variable for providing a unique id for an expense report.

For example, consider the facts with the given intervals:

$$\mathit{salary}(s1, 500)@0. \qquad \mathit{salary}(s2, 500)@120.$$
$$\mathit{claim}(c1, 50)@96. \qquad \mathit{noResp}(c1)@[96, 120].$$

Then, we retrieve the following (final) facts

$$\begin{aligned} &expense(s1, 500)@0. && expense(s2, 500)@120. \\ &expense(c1, 50)@120. \\ &expenses(500)@[0, 120). && expenses(1050)@[120, \infty). \\ &expense(-1, 500)@(0, 120]. && expense(-1, 1050)@(120, \infty). \\ &report(r1, 500)@[0, 120). && report(r2, 1050)@[120, \infty). \end{aligned}$$

and the intermediary recursive results $expenses(500)@0$, $expenses(550)@120$ by Rule 3, and by Rule 4 $expense(-1, 550)@(120, \infty]$ and $expenses(550)@(120, \infty]$.

Based on existing benchmark tools and our observations in Example 4.1, we recognize the following fundamental requirements of a temporal benchmark generator:

1. *Full recursion*: The benchmark generator should be able to build rules that explore full recursion for a fixed timepoint (i.e., default Datalog recursion) as well as over the temporal domain (i.e., temporal recursion – a set of recursive rules that contains a temporal operator).
2. *Temporal Operators*: The benchmark generator should at least support the temporal operators of DatalogMTL.
3. *Aggregation*: The benchmark generator should be able to calculate the aggregation per timepoint. Other forms of temporal aggregation are not required as those can be simulated by using additional temporal operators (see Chapter 3).
4. *Existential Quantification*: The benchmark generator should be able to handle existential quantified rules, at least supporting weakly-acyclic programs under the uniform semantics.
5. *Extensibility*: The benchmark generator should be able to be extended with additional features. This includes additional rule types (e.g., detection of temporal trends, temporal constraints, etc.) as well as exchangeable components. For example, to generate real-world like scenarios, one has to have the option to replace the data generation by statistics of the underlying scenario.

While the first four requirements allow one to generate queries that cover ones day-to-day programs and address the shortcomings of existing solutions, the fifth requirement ensures that the generator can be extended in the event that additional features are required, rather than being a generator for a single setting like existing solutions.

4.2 The Core

In this section, we present the core of our benchmark generator. We first provide a general overview and then describe each phase of the generation algorithm in detail.

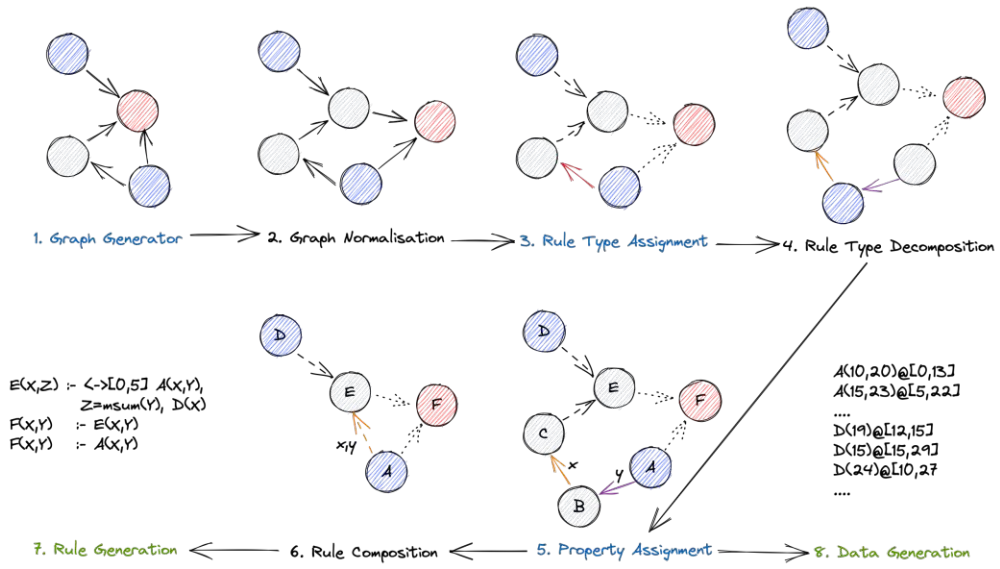


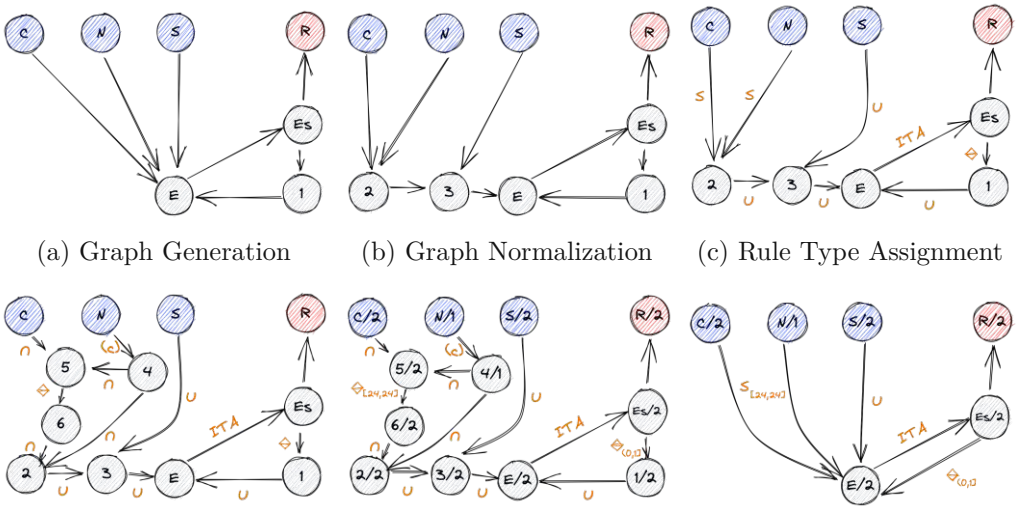
Figure 4.1: Overview of the phases of the benchmark generator

4.2.1 Overview

The core of our generator adapts the concept of a dependency graph of Datalog programs for the generation process, which we call a *generative dependency graph*. A generative dependency graph is a graph whose nodes represent the predicate names over which our programs are formulated, and edges represent dependencies between predicate names. These dependencies represent templates for rules, where the tail predicate appears in a body atom, and the head predicate appears in the head atom. Note that the plain generative dependency graph deliberately does not contain the arities of predicates (hence we speak of predicate names), because selecting them is an important part of generation. Hence, this generative dependency graph is augmented in two ways: (i) each edge is annotated by a *rule type*, such as type $\diamond$, and (ii) each node and edge are annotated by *properties*. For an edge of rule type $\diamond$, the property is an interval (e.g., $[3, 4]$). For a node, a property is the arity of the predicate.

Definition 4.2.1. A *generative dependency graph* is a directed graph where nodes are predicate names. An edge may have a *rule type*. A node or edge may have *properties*, which are pairs of *property name* and *property value*.

Note that the dependency graphs play a similar role in (recursive) data processing systems as logical query plans play in traditional database systems and more generally, data-flow graphs in many parts of Data Engineering: they are abstract representations of the query. In a generator, apart from being a natural data structure for providing the query, they allow to generate data by following this data-flow graph. Hence, the more a given dependency graph is augmented with types and properties, the more restricted the generated programs represented by it are. A fully augmented dependency graph (i.e., all



(a) Graph Generation (b) Graph Normalization (c) Rule Type Assignment

(d) Rule Type Decomposition (e) Property Assignment (f) Rule Composition

$$
\begin{aligned}
\mathit{expense}(\mathrm{Id}, \mathrm{P}) &\leftarrow \mathit{salary}(\mathrm{Id}, \mathrm{P}) \\
\mathit{expense}(\mathrm{Id}, \mathrm{F}) &\leftarrow \mathit{noResp}(\mathrm{Id}) \mathcal{S}_{[24,24]}\, \mathit{claim}(\mathrm{Id}, \mathrm{F}) \\
\mathit{expenses}(-1, \mathrm{T}) &\leftarrow \mathit{expense}(\mathrm{Id}, \mathrm{P}), \mathrm{T} = \mathit{msum}(\mathrm{P}) \\
\mathit{expense}(-1, \mathrm{T}) &\leftarrow \diamondminus_{(0,1]}\, \mathit{expenses}(\_, \mathrm{T}) \\
\mathit{report}(\mathrm{Id}, \mathrm{T}) &\leftarrow \mathit{expenses}(\_, \mathrm{T})
\end{aligned}
$$

(g) Rule Generation

$\mathit{Salary}(s1, 500)@0.$
$\mathit{Salary}(s2, 500)@120.$
$\mathit{Claim}(c1, 50)@96.$
$\mathit{NoResp}(c1)@[96, 120].$

(h) Data Generation

Figure 4.2: The state of the graph of Example 4.1 after each phase.

possible types and properties are set) precisely individuates one single program/query. That is, a generative dependency graph can, in principle, be instantiated by arbitrary DatalogMTL rules that follow the above-mentioned patterns. However, in the context of our generator, we need more fine-grained control over this generation, allowing to modify the generative dependency graph along the generation process.

Phases. Procedurally, the generator revolves around eight phases, visualized in Figure 4.1. The eight-phase design was conceived to reflect real-world instances to an arbitrary degree. We grouped the related tasks together so that each user-input (phases 1 – the generation of the plain dependency graph, 3 – the selection of rule types, and 5 – the selection of properties) is followed by a required transformation of the graph (phases 2, 4, and 6). For real-world use cases, one provides (full) input in all three input phases to specify the query. Yet, one often wishes to generalize to make sure to cover a broader range of settings by omitting some of the input in phases 1, 3, and 5.

For this, all eight phases are designed in an extensible way, so that new modules can interact with them and support the (partial) pre-configuration of certain phases (marked

in blue in Figure 4.1) by the user to support interaction during the generation process. This means that the user can replace the graph generation with a custom graph (phase 1), pre-assign specific rule types to edges in a graph (phase 3) and choose specific properties (phase 5). The phases 7 and 8 produce the actual output, i.e., the query (program) and the data. We now describe the phases in detail:

Example 4.2. Figure 4.2 shows the result after each phase for the generation of the running example. We will reference to this example in the following subsections.

1. **Graph Generation**. The first step is to build a directed graph. Such graph can be generated by either one of the many existing graph generation tools, or our generation tool that we provide together with the benchmark generator. It offers the possibility to control or hand-pick the desired amount or form of recursion, especially relevant to carefully select the underlying graph structure for targeted test scenarios.

2. **Graph Normalization**. This step decomposes the graph in such a way that each node has at most two incoming edges. This simplifies the following steps without loss of generality, as we shall see.

3. **Rule Type Assignment**. So far, the graph just contains nodes and edges, without any associated semantics. In this step we assign to each edge a type, i.e., we decide whether it represents a $\boxminus$-, $\diamondminus$-based, etc. rule.

4. **Rule Type Decomposition**. We decompose edges into simplified operations. For example, we convert a *since* edge to a $\diamondminus$, intersections, and a closing edge.

5. **Property Assignment**. In this crucial step, the interplay between the edges and nodes is realized. We assign the arity of the nodes and decide on the attributes, e.g., the intervals of the temporal operators.

6. **Rule Composition**. After having assigned the properties to the nodes, the next goal is to generate the rules. As the dependency graph is an internal representation, to make rules executable, we may have to merge (i.e., compose) some of them, to facilitate the direct conversion of the graph into a query syntax that fits the target platform.

7. **Rule Generation**. In the last step of the rule generation, the result of the rule composition is used to generate the query. For the moment, we support queries for our Vadalog system [BSG18], non-recursive queries for PostgresSQL, as well as a restricted set of queries for QuestDB [Que21], a state-of-the-art time-series database.

8. **Data Generation.** As we have seen, the availability of data is a fundamental ingredient in a benchmark. Such data should be (i) of types that are supported by

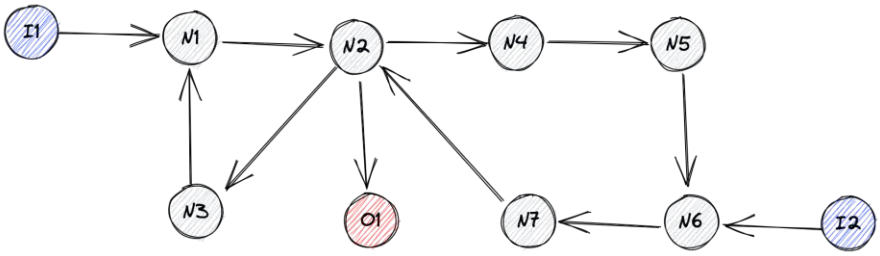


Figure 4.3: Result of a generated graph.

the target systems and, (ii) fitting with the generated rules so that they produce non-empty results.

The flexibility of the system, given to the user by the possibility to choose between a generated graph of our tool or decide for any other (partially assigned) initial graph, combined with the non-triviality posed by the undefined (and potentially large) number of operations causes a series of technical challenges. This includes controlling the difficulty of recursion during graph generation, assigning the atom (node) arity, which cannot be chosen arbitrarily but is limited by the concrete graph structure and rule types, and generating data that is appropriate for the given flexible graph structure, all of which have to be solved while supporting extensibility (requirement 5).

Example 4.3. For example due to the *flexibility*, a user who generates a benchmark can provide as input a custom graph (i.e., skip phase 1), with partial assigned edges (i.e., skip (some) assignments of phase 3 such as providing only the information that an edge is temporal but not the temporal operation) with some of them containing specific properties (i.e., skip (some) assignments of phase 5 such as providing no intervals to temporal operators).

4.2.2 Graph Generation

This phase generates the graph that is then the input of the subsequent phases. The structure of the graph is essential for the support of different use cases. In principle, any off-the-shelf graph generation tool could be adopted here, however, let us discuss the fundamental required characteristics of an optimally tunable graph generator:

- *Number of Nodes.* The total number of nodes to decide how large the benchmark should be.
- *Number of Inputs.* The number of nodes that contain no incoming edges. This is required to interweave the generator with possible other programs and to know for which nodes data should be generated.

- *Number of Outputs.* The number of nodes without outgoing edges. This is required to interweave the generator with other possible programs and to know which nodes have to be considered for generating queries.
- *Percentage of nodes with multiple incoming edges.* Depending on the number of incoming edges, different rules will be generated. For example, rules (1), (2), (3), and union require multiple incoming edges, while (4), (5), $\diamond$, $\oplus$, and linear rules require a single incoming edge.
- *Percentage of recursive edges.* In order to support recursive queries, we have to control the number of recursive edges in the graph, i.e., those edges whose presence in the graph determine a cycle.
- *Recursive complexity.* Recursion can be of increasing complexity levels, starting with trivial chordless cycles. With this parameter, the user can control recursion by adjusting the number of connected components in which the nodes, having indegree higher than one, are involved.

To the best of our knowledge, a graph generator that supports all these requirements together is not present and is therefore an essential ingredient of our benchmark tool. The parameters we use from hereinafter are either chosen from a Gaussian distribution by providing mean and variance or by providing a value between zero and one.

Example 4.4. Figure 4.3 shows such a generated graph with 10 nodes (thereof 2 inputs, 1 output), 40% multi-edges, 20% recursive edges and recursive complexity of 0.6. Figure 4.2a shows one possible graph for the running example.

Our generator uses a multi-step build process. In the first step, we partition the graph into strongly connected components based on the number of nodes with multiple incoming edges. This step depends on the recursive complexity, which, as we have seen, selects how many such nodes are put into a single strongly connected component. Then we connect the nodes inside the strongly connected component ensuring that the number of recursive edges is met. Finally, we connect the strongly connected components to form a direct acyclic graph from the input nodes via the strongly connected components to a single output node. The other outputs are given by selecting a random node in the graph and adding a path to the output node.

Example 4.5. The parameter of recursive complexity takes a value between 0 and 1. In case of 0 the graph is chordless, in case of 1 it introduces the maximum number of chords possible based on the other parameters (i.e., in “worst case” a fully connected graph).

4.2.3 Graph Normalization

The graph normalization phase has the goal of simplifying the graph to a normalized format. In particular, we simplify the graph in such a way that (i) there are at most two

(a) Before Normalization (b) After Normalization

Figure 4.4: Normalization of edges

incoming edges to a single node and (ii) all incoming edges to a single node belong to the same type. While the former is independent of the user's input, simplifies the handling of later phases, and has to be applied for any generated graph, the latter is only required as we support the pre-assignment of properties for the next phases to ensure flexibility of the generation process.

Example 4.6. We visualize (i) in Figure 4.2b and (ii) in Figure 4.4, where we combine the introduction of the middle node of a boxminus (⊟) edge and the propagation of the intersection information, noted via the mathematical symbol (∩).

We define the normalization process as follows: Let x be the number of incoming edges to a node n from nodes m_i, with $0 \leq i < x$, and let us denote with e_i the incoming edges to n, i.e., $e_i = (m_i, n)$. Then, we compute the normalization of the graph as follows:

Conversion of single-edge rules. We first start with (ii) where we first convert all rules requiring a single incoming edge and then check for consistency. In case $x \geq 2$ and there is an edge e_i whose type is none of *generic* (i.e., denoting it is not assigned yet), *intersection*, *union*, *since*, or *until* then create a node o, change the edge e_i to (m_i, o) and add an additional generic edge (o, n).

Type propagation. We now ensure that all incoming edges to a node belong to the same type. For this, we ensure first that there is no conflicting assignment, i.e., (a) there are not edges e_i and e_j for $0 \leq i, j < x$, so that the edge type of e_i is not *generic*, e_j is not *generic* and the types are different, and (b) in case of *since* and *until* the number of incoming edges is exactly two. In case there is such a conflict, then the user provided an indistinguishable dependency graph, i.e., we do not know which edges should be considered first in the evaluation.

In case the user provided an edge type at least for one edge (i.e., for some e_i the type is not *generic*) we propagate this type to all incoming edges to ensure that all edges belong to the same type.

Middle Node Insertions. In the last step, we focus on (i) and transform nodes with $x > 2$ by inserting nodes so that each node has at most two incoming edges. We create $x - 2$ middle nodes o_i. We connect the first two edges e_1, e_2 to the first middle node o_1, the last middle node o_{x-2} and the last edge e_x to n, and each remaining middle node o_{i-1} and the edge e_{i+1} to the next middle node o_i.

4.2.4 Rule Type Assignment

In Section 4.2.2 we already discussed that certain rules require a different number of incoming edges. In this section, we propose a two-phase rule assignment process. In the first phase, we assign a category to each rule marked as *generic*. In a second phase, we select one specific category type for each edge. This two-phased process gives the user the possibility to fine-tune the distribution of rule types used in the generated benchmark. We structure the rule categories/types as follows for single incoming edges:

- *Elemental*: Linear
- *Temporal*: BoxMinus, DiamondMinus, BoxPlus, DiamondPlus

and for multiple incoming edges:

- *Elemental*: Intersection, Union
- *Temporal*: Since, Until

That is, we allow the user to provide a percentage for each category (e.g., Elemental (Single) or Temporal (Multi)) and the sum of the percentages of each category has to sum up to 100% for single, for multiple incoming edges, and for the types in each category.

In detail, the assignment process is implemented as follows: First, we count the number of edges per type provided by the user and the total number of edges. We then assign according to the computed statistics to each *generic* edge the selected category, and finally assign per category the exact type.

Example 4.7. In Figure 4.2c we see such an assignment for the running example with a configuration of 100% for Since, Union, Linear, DiamondMinus and ITA (see Section 4.3) and the categories evenly distributed for single incoming edges and 33% (resp. 66%) for temporal (elemental) for multiple incoming edges.

4.2.5 Rule Type Decomposition

In the previous step, we assigned a rule type to each edge. While we provide the user with a diverse set of rule types for the assignment, not all of them are made up of a single operation and can be represented by a composition of rules.

The goal of this phase is the conversion of such rules (edges in the dependency graph) into simplified components to simplify the data generation as well as rule generation for different platforms. In the core module, we have to consider the *until* and *since* rules, which can be decomposed on an interval-level to diamond ($\diamondplus$ resp. $\diamondminus$), intersections and a closing-rule (a rule, that just closes open non-infinite intervals) [WCGKK19]. Note, that when this decomposition is applied on a fact that contains multiple intervals, then the

decomposition satisfies facts for more intervals than allowed. Yet, we explicitly favored this decomposition as it simplifies the following generation steps from an implementation perspective (i.e., one does not have to handle a complex join that requires for each pair of intervals to (i) join, (ii) apply the diamond operator and (iii) join again) under the presumption that this simplification should not have a large impact on the data generation process.

Example 4.8. The decomposition of the *since* rule is visible in Figure 4.2d, the *until* rule is analogous by replacing $\diamondminus$ with $\boxminus$.

4.2.6 Property Assignment

We now have established a graph that is reduced to edges that contain unique operations, i.e., a single job per node (e.g., computing the intersection of incoming edges, or shifting the interval of the incoming data). While this provides the general structure of the rules in the program, we still miss information to generate a final program. For example, we still have to choose the arity of the nodes[1], the intervals of the temporal operators, and so on. In this phase, we exactly select those properties.

Example 4.9. In Figure 4.2e, we show the graph of the running example with assigned arities and intervals.

The property selection is built around the arity selection of the nodes by a multi-phased process. That is the case, as some properties depend on the arity, which cannot be freely chosen per node as the following proposition shows.

Proposition 4.1. *Let S be a strongly connected component at the beginning of phase 5. If there is no intersection edge in S, then the arity for all nodes in S is the same.*

Proof. Assume there is a strongly connected component (SCC) with n nodes and each node has some arbitrary arity x_i. We show that all nodes have arity $x = min(x_i)$. By the definition of the dependency format, each node has either two incoming edges (union or intersection) or one incoming edge (linear, temporal operators) and at least one incoming and outgoing edge must be connected to a node in the same SCC, otherwise the SCC condition would be violated. We now chose a node y such that the outgoing edge reaches a node z, such that $x_y < x_z$. Node z has either a single incoming edge (the edge of node y or multiple incoming edges). By definition of DatalogMTL, each rule is safe (i.e., there is no support of existential quantification) and duplicates of variables are forbidden by our assumption. Hence, in case it has a single incoming edge, the edge cannot create additional terms and hence the arity of the node has to be updated to x_y. In case, there are multiple incoming edges, then by restriction of the theorem it must

[1] We assume that a single body term maps exactly to a single head term. This represents no restriction to the generator, as multiple occurrences can be simulated by using intersection of newly created auxiliary predicates.

Algorithm 4.1: Algorithm for computing arity of nodes

Input: Dependency Graph G
Output: Dependency Graph G, each node assigned its arity

```
1  functions ← [maxArity, minArity, infArity, intersectionArity,
     randomArity ];
2  components ← getSCCs(G);
3  while components.length > 0 do
4  |   component ← getNextSCC(components);
5  |   setInitialArity(components.nodes);
6  |   hasChanged ← true;
7  |   while hasChanged = true do
8  |   |   hasChanged ← false;
9  |   |   for function in functions do
10 |   |   |   hasChanged ← function(component);
11 |   |   |   if hasChanged = true then
12 |   |   |   |   goto Line 7;
13 return G;
```

be a union edge. As all incoming union edges must have the same arity, by the same argument, the arity has to be updated to x_y. This process can be repeated for each node in the dependency graph until there is no further change. That is, at the end all nodes have arity $x = min(x_i)$. □

Hence, we structure our process into a pre-arity-selection phase, an arity-selection phase, and a post-arity-selection phase, which we discuss in the following.

Pre-Arity and Post-Arity Phases Before beginning the arity property selection procedure, the pre-arity phase ensures a preliminary graph structure (e.g., that input and output types are assigned to the corresponding nodes). We assign attributes that depend on the arity in the post-arity phase. One such example is the order of terms (i.e., which term of the source node is passed to which term in the target node), another is the modification for existential quantification (see Section 4.4). We also include activities like assigning parameters that are independent of arity, such as the interval of temporal operators, which can be chosen in either the pre-arity or post-arity phase.

Example 4.10. The results of the property selection of the running example are given in Figure 4.2e. Note that we have added intervals for the temporal operators (pre-arity), the arity of the nodes (arity phase) and the existential quantified variable in the output node (post-arity).

Arity-Selection Phase The goal of this phase is to assign arity dependent properties. This includes the arity of the node, as well as the number of joining and non-joining

Algorithm 4.2: Algorithm for maximum arity assignment

Input: Dependency Graph Component C
Output: True, if the component has been updated, else false

```
1 for node in C.nodes do
2     maxOp ← sum if node.type = Isection else min;
3     newMaxA ← node.inEdge.maxOp(source.maxA);
4     if newMaxA < node.maxA then
5         node.maxA ← newMaxA;
6         return true;
7 return false;
```

terms for intersections. As Proposition 4.1 has shown, the variance of the nodes' arity inside a single connected component is restricted by the operations. In order to compute the arity, we propose in Algorithm 4.1 a back-propagating approach that manages the current possible minimum and maximum arity of each node in the graph starting with the output nodes. The advantage of a back-propagating approach is the gained flexibility with the maximum arity, as the arity only increases along the path. In case a specific input arity is required, one can run a forward propagation of the input arity to the other nodes to restrict the arity of following nodes.

Algorithm 4.1 takes as input the current dependency graph and returns a modified dependency graph that contains for each node an assigned arity according to the algorithm. The algorithm computes in Line 2 an ordered list of strongly connected components (starting with the output nodes) and for each strongly connected component we initialize the nodes (Line 5) with the minimum and maximum arities (i.e., some random arity for output nodes, the pair (MIN_ARITY, ∞) for nodes that receive a minimum arity by an already handled strongly connected component and for all other nodes the pair $(0,\infty)$[2]) and execute a list of functions until nothing has changed (6-12). At the end, all arity-dependent parameters are assigned. In the following we explain the *functions* in detail:

Maximum Arity Propagation (Algorithm 4.2). This algorithm propagates the maximum arity to the *following* node in the same strongly connected component. The new maximum arity is the sum of the maximum arities of the source nodes for intersections and the minimal maximum arity of the source nodes for all other nodes. The maximum arity is only updated in case the new maximum arity is lower.

Minimum Arity Propagation (Algorithm 4.3). This algorithm propagates the minimum arity to the *previous* nodes in the same strongly connected component. The new minimum arity is the maximal minimum arity of the target nodes. In case the target node is connected via an intersection edge, the minimum arity is defined as the difference

[2]We followed the approach to minimize the arity of each node. Optionally, we increase the minimum arity between strongly connected components.

Algorithm 4.3: Algorithm for minimum arity assignment

Input: Dependency Graph Component C
Output: True, if the component has been updated, else false

```
1  for node in C.nodes do
2      for out in node.outEdges do
3          if out.type = Isection then
4              newMinA ← out.target.minA - out.target.
                 inEdge.otherEdge(out).source.maxA);
5          else
6              newMinA ← out.target.minA;
7          if newMinA > node.minA then
8              node.minA ← newMinA;
9              return true;
10 return false;
```

between the minimum arity of the target node minus the maximum arity of the other intersection edge. The minimum arity is only updated in case the new minimum arity is higher.

Infinite Arity Assignment. The goal of this assignment is to pin down the arity of a specific node. For this, we choose a node that is connected with an outgoing edge to a different SCC (i.e., a component we already have a minimum arity from that component), its maximum arity is infinite and has the highest minimum arity. In case such a node exists, we set its maximum arity to the minimum arity. This influences the propagation of the maximum arity through the components.

Intersection Arity Assignment (Algorithm 4.4). This algorithm assigns the number of join terms and non-join terms to an intersection node and fixes the arity of such an intersection. First, we compute the number of join terms (Lines 6-10), where it is essential to consider the minimum number of required non-join terms to fulfil the chosen arity (Lines 4-5). Then, we choose the number of non-join terms, where the minimum (resp. maximum) number of the one source node's arity depends on the maximum (resp. minimum) arity of the other node's arity. Note that in addition to the presented algorithm we internally use the concept of reference edges that ensure that the arity between edges is synchronized, which is important for the support of the since/until operator.

Example 4.11. Consider a node with an arity of up to 5 (in Line 4-5) and two incoming edges, the first with maximum arity 2 and the second with maximum arity 3 (yielding *minMaxArity* of 2 and *maxMaxArity* of 3). Then the algorithm computes the following values, where the final value of the ranges can be determined by choosing jT (e.g., if jT in range $0-2$ has value 1 and $noJT$ has a range of $3-1$, then the value in this range must be 2).

Algorithm 4.4: Algorithm for intersection arity assignment

Input: Dependency Graph Component C
Output: True, if the component has been updated, else false

```
1  for n in C.nodes.filter(type = Isection) do
2      if n.propertiesAssigned then
3          continue;
4      n.minA ← nextInt(n.minA, n.maxA);
5      n.maxA ← n.minA;
6      minMaxArity ← min(n.inEdge[0].source.maxA, n.inEdge[1].source.maxA);
7      maxMaxArity ← max(n.inEdge[0].source.maxA, n.inEdge[1].source.maxA);
8      minNoJT ← max(0, n.minA − maxMaxArity);
9      maxJT ←min(minMaxArity − minNoJT, n.minA);
10     jT ← nextInt(0, maxJT));
11     noJT ← n.minA − jT;
12     maxE1 ←min(noJT, n.inEdge[0].source.maxA − jT);
13     maxE2 ←min(noJT, n.inEdge[1].source.maxA − jT);
14     minE1 ←max(0, noJT − maxE2);
15     noJT1 ←max(minE1, nextInt(0, maxE1));
16     noJT2 ← noJT − noJT1;
17     updateJoinTerms(n.inEdge, jT, noJT0, noJT1);
18     return true;
19 return false;
```

n.minA	*minNoJt*	*maxJt*	*jT*	*noJT*	*maxE1*	*maxE2*	*minE1*
5	2	0	0	5	2	3	2
4	1	1	0-1	4-3	2-1	3-2	1
3	0	2	0-2	3-1	2-0	3-1	0
2	0	2	0-2	2-0	2-0	2-0	0
1	0	1	0-1	1-0	1-0	1-0	0
0	0	0	0	0	0	0	0

Random Node Arity Assignment. In case there is no change in the previous steps and there are nodes where its maximum arity is higher than its minimum arity, then we pick such a node with the highest minimum arity and set its maximum arity to a random value between the minimum and maximum arity. This ensures that in case no other assignment works, we can continue the assignment.

Example 4.12. For the running example, we start by computing the SCCs which are the set of nodes (E, 1, and E_s) and an own SCC for each node not in the set. Then we continue with the assignment of the initial minimum and maximum arity of the nodes. The report (output) node gets the minimum and maximum arity 1^a (short

$(1,1)$) and all other nodes the initial arity $(0,\infty)$. The minimum arity is transferred to E_s and (optionally) increased by 1 leading to arity $(2,\infty)$ and by the minimum arity propagation, the other entries get the same minimum arity. By the infinite arity assignment, the chosen node gets the min/max value of $(2,2)$, which is propagated to the other nodes by the maximum arity assignment. This also fixes the aggregation parameters (see Section 4.3.2) to the aggregation term, one group-by term, and no contributor or other terms. By the same logic, this value is propagated to the SCCs 3, S, and 2, as well as for 5 and N later in the process. What remains are the intersections, where we have one join term, and one (resp. no) additional terms for 6 (4). The same is forced by the given constraints for node 4 to the nodes C and 4.

[a] The final arity 2 is derived by introducing an existential quantified variable (see Section 4.4).

4.2.7 Rule Composition

We need to target many different platforms for rule execution, each having different languages, dialects, and characteristics. Thanks to the internal representation, our approach can be translated into different languages. In fact, while for Datalog the translation is straightforward, other target platforms require more care. In any case, we proceed by composing the rules, which means that we merge and transform the nodes to a compatible graph, in such a way that we can simply iterate over the graph for rule generation. For example, when targeting a relational or SQL-like database (e.g., QuestDB), we convert the graph to nodes and edges following a SELECT-FROM-WHERE structure in case the current graph (query) is compatible with the target platform, or for Datalog we revert the rule type decomposition step.

Example 4.13. Consider the running example (Figure 4.2f) again. There we transformed the graph to the minimal required rules for Datalog by removing auxiliary nodes in the final query.

4.2.8 Rule Generation

The last step is the actual generation of the query (program). In this phase, the goal is to convert the nodes and edges to the query format of the platform. Algorithm 4.5 highlights the process for Datalog-based systems. In short, we iterate over each node, considering the incoming edges as individual body atoms and the node as head of the rule. Depending on the Datalog-specific language, certain operations may be output differently, e.g., the symbol of the temporal operator, how the interval is specified, and so on. We consider three options to insert operators, one before the atom (e.g., for the temporal operators), one after the atom (e.g., for aggregations), and one between atoms (e.g., for joins).

Example 4.14. Consider Figure 4.2g. It shows a generated rule set for the running example.

Algorithm 4.5: Algorithm for generating rules

```
   Input: Dependency Graph G
   Output: The program P
 1 P ← "";
 2 for n in G.nodes do
 3 |   if n.type is Multi then
 4 |   |   P += n.getAtom() + ":-";
 5 |   |   P += e.source[0].getAtom();
 6 |   |   P += e.getPossibleJoinOperator();
 7 |   |   P += e.source[1].getAtom();
 8 |   else
 9 |   |   for e in n.inEdge do
10 |   |   |   P += n.getAtom() + ":-";
11 |   |   |   P += e.getPossiblePreOperator();
12 |   |   |   P += e.source.getAtom();
13 |   |   |   P += e.getPossiblePostOperator();
14 return P;
```

4.2.9 Data Generation

So far we concentrated on the generation of rules. Yet, one key ingredient for generating benchmarks is missing, namely the data. Hence, this phase is about this final component. Data generation per se is a highly critical task as it is important that every component of the graph contains data such that the following benchmarking rules have an effective impact on the performance and do not produce empty results. This is non-trivial as one has not only to deal with joins, but also with recursion and temporal data.

If one were to choose a forward-propagating approach (i.e., choosing random variables for the input atoms (nodes)), one has to carefully select those tuples, such that whenever an intersection (join) of atoms is required, the time of some atoms overlaps. As one can see, one has to anticipate how the data will flow and adapt the input generation accordingly, which includes recursive structures, which may shift the time forward or backward, aggregations which results depend on the number of tuples in the joins, and so on.

In order to overcome that issue of anticipating what follows, we propose a back-propagating approach. The goal of this approach is to start with a seeding data set for the output atoms and back-propagate this information to its predecessors. We want to emphasize the word "seeding", as an application of a forward propagation after the backpropagation may yield additional, non-covered values by the back-propagation[3].

[3] As the generation of data is complex, one can use the seeding as well for getting statistical information of the input nodes which can be used to produce further input data. We have not implemented this but consider it as a viable option for future versions.

The data generation evolves over three critical components in the core model dealing with temporal operators: (a) the generated output intervals must have a minimum size, such that the temporal operators can be applied, (b) in cyclic graphs they cause infinite chains, and (c) one has to decide which recursive generated data is kept in the input as not all data is required. We discuss these three points in the following.

Minimum Interval Size. Temporal operators produce new intervals. When back-propagating the output intervals, we apply the opposite mathematical operations. That is, if we subtract some intervals, then the produced interval may be of negative size and is discarded. Hence, it is important to produce an interval of minimum size. In order to generate at least one valid interval, we need to know the minimal interval length of the output intervals, which we solve by forward-propagating the temporal operators. That is, we start by the input nodes with an interval of $\langle 0, 0 \rangle$. For each node, we compute the shortest interval, that is, we pass the interval along each edge until there is no further change. We compute the target node's interval length (i) for temporal operators by applying the temporal operator to the current interval of the source node, (ii) for intersection and union nodes, we choose the interval with the smaller length, and (iii) for all other edges we copy the interval. In case we have cycles with negative length, we restrict the minimum length of the interval in the SCC to zero.

Example 4.15. In case of the running example, the minimum interval length is zero, as there is a direct path from the output to one input.

Infinite Intervals. Infinite chains are created by a recursive application of temporal operators, which colloquially speaking shifts the interval always by an amount n forward or backward in the timeline. In order to prohibit infinite cycles, the user can apply a likelihood that defines the probability of how likely the derived fact comes from a previous cycle. So, per each iteration, we will lose some of the facts until no additional facts are back-propagated.

Minimum Datasets. While we back-propagate the intervals, all intervals from the previous recursive rounds have been stored. Providing all of those data to the next SCC is not useful, as the temporal operators would produce data that is already in the set of existing data and hence the recursive application of temporal rules cannot be measured (i.e., each application of the rule produces an existing interval in the dataset). In order to remove such data, we apply a forward propagation to the derived facts (starting with the facts derived latest) to remove data that is already generated by the recursive application. As also the forward propagation can cause infinite intervals, we limit the produced facts to existing data generated during the back-propagation step and discard any other intervals. That is the case, as it is only important to remove the existing data from previous recursive rounds.

Example 4.16. One trivial setting of the running example (Figure 4.2h) for seeding data of the output may be $R(r1, 500)@0$, $R(r2, 500)@120$ and $R(r3, 50)@[96, 120]$ with a really high probability that facts will not be recursively iterated via the cycle.

4.2.10 Time and Space Complexity

In this section, we shortly discuss the time and space complexity of the generator. Apart from data generation, the generator is polynomial in time and the space requirement is linear in the graph size. For data generation, it is in fact primarily the user parameter "temporal recursion percentage" (which determines how likely a given fact will recursively produce another fact) that determines the complexity. For low values, it is polynomial in the graph size, if a user chooses 100%, the system will produce data until manual cutoff. This gives the user full control over the complexity here.

4.3 Aggregation Module

In this section, we present our extension to the core module which focuses on aggregation. As discussed in Section 2.3, there are various kinds of temporal aggregations we have to consider in the benchmark generation process. We first start to describe how this module integrates with the core, and then introduce the characteristics of this module, e.g., property assignment of aggregations, data generation, and so on.

4.3.1 Integration with the Core

We developed the core module in such a way that additional tasks from other modules can be registered with a priority (managing the execution order of the tasks in the corresponding phase). This allows one to exactly integrate the new tasks where they are required by the module. In the following, we explain the additions of this module.

- In the phase *rule type assignment*, we add an additional category *aggregation* with the three main aggregation types, i.e., ITA, MWTA, and STA.
- In the phase *rule type decomposition*, we add a decomposition of MWTA and STA as described in Section 3.1 to its individual components. In summary, this leads to additional ITA and triangle-up edges (for STA aggregation) in the resulting dependency graph of this phase.
- In the phase *property assignment*, we have to consider the properties of the new edges. Especially ITA has an influence on the arity, which we discuss in Section 4.3.2.
- In the phase *data generation*, we have to be aware of the new edge types as they influence the possible intervals that are produced.

Algorithm 4.6: Algorithm for aggregation arity assignment

Input: Dependency Graph Component C
Output: True, if the component has been updated, else false

```
1  edge ← C.edges.filter(type = Aggregation)
     .filter(hasUnassignedAggrProps).selectRandom ;
2  if edge = NULL then
3  |  return false;
4  grBys ← numberOfGroupBys() ;
5  grBys ← min(edge.target.maxA − 1, grBys);
6  grBys ← max(edge.target.minA − 1, grBys);
7  contrib ← numberOfContributors() ;
8  sub ← 1 + grBys;
9  contrib ← min(edge.source.maxA − sub, contrib);
10 sub ← sub + contrib;
11 others ← numberOfOthers() ;
12 others ← min(edge.source.maxA − sub, others);
13 others ← max(edge.source.minA − sub, others);
14 updateAggregationTerms(edge, grBys, contrib, others);
15 return true;
```

4.3.2 Adaption of the Arity Algorithm

As mentioned in Section 4.3.1 aggregation has an influence on the arity. That is the case, as the syntax of (monotonic) aggregation in Datalog defines different groups of terms, as introduced in Section 3.1:

- *Group-by terms.* These terms are forwarded to the head to group the aggregation.
- *Aggregation term.* This term is the aggregate. For count, such a term is not required, but at least some other term is required so that the count is useful (i.e., does not return 1 per group-by key).
- *Contributor terms.* To support monotonic aggregation, only the maximum value per contributor is considered in the aggregation. These terms are only allowed to be part of the body, as only the maximum aggregation term of these contributors is part of the aggregation result.
- *Other terms.* These terms have no direct meaning in the aggregation but allow to have multiple data values that may influence the aggregation result.

As one can see, the arity of an aggregation depends on multiple parameters. We decided to add the arity assignment directly before the `intersectionArity` function of Algorithm 4.1. This gives us enough flexibility to select the different aggregation parameters where possible, before narrowing down the amount with the intersections.

Algorithm 4.6 highlights the selection process of the number of group-by terms and contributor terms. First, we decide on the number of group-by terms. The range of group-by parameters is selected by a Gaussian distribution (Line 4) based on user-defined parameters and is limited by the minimum and maximum arity of the target (Lines 5-6). We then add contributors and other terms in a similar way (Lines 7 and 11). These parameters depend on the minimum and maximum arity of the source node (Lines 9 and 12-13), where they have to sum up together with the group-by terms to reach the minimum arity of the source node. We assign the chosen parameters to the edge and the received arity to the nodes (Line 14). We then continue with the distribution of the maximum and minimum values from the previous steps (discussed in Section 4.2.6), before considering the next aggregation term, as this fixes the arity of dependent nodes and other aggregations are depending on the propagated arity information.

4.3.3 Data Generation

In the core module we discussed the characteristics of the temporal operators where we introduced a back-propagating algorithm for data generation. As both newly introduced edge types have an impact on the generated intervals, we describe the changes in the following.

Aggregation. In the back-propagation step, we receive as value the output of the aggregation and have to generate the input values. As discussed in the previous section, there are different terms that influence aggregation. Hence, the input values may vary in multiple dimensions, and we have to consider that when generating the data. That is, we split a fact by (i) the time interval, (ii) the aggregation value (and other terms), or (iii) its contributor terms to create different input facts to the aggregation. We handle the support of the three dimensions as follows:

1. We start by splitting the aggregation interval into n buckets. A bucket contains the aggregation values for a specific sub-interval of the output interval. For the creation of the buckets, we create $n-1$ random dividers between the interval range and use these dividers to create the boundaries of the sub-intervals. (This creates the possible splitting along the time (i))

2. For each bucket (starting with the first one):

   a) We compute the number of facts in the bucket.

   b) Randomly move some facts from the previous bucket to the current bucket (to support different input interval lengths of the facts).

   c) Then, we fill up part of the remaining bucket with randomly generated facts (keeping the aggregation result in mind).

   d) And finally, we fill up the remaining part with contributor values by selecting a specific fact of the bucket and modifying the aggregation value.

3. Finally, we merge equal facts that are adjacent in the buckets.

In order to avoid any possible dependencies that influence the aggregation negatively, we: (i) compute per temporal-interval the aggregation only once, and (ii) during the removal of recursive generated temporal intervals (i.e., during the forward-propagating step), we map the aggregation to the value in storage instead of computing the aggregation again to improve performance.

Triangle-Operator. The time granularity operator extends the interval to a certain time-range. In the back-propagation step, the input interval (i.e., the interval at the target node of the edge) has to match exactly the granularity definition (e.g., if the granularity is month, the interval range has to match for example 1 Jan to 31 Mar, 10 Jan to 20 Mar would not be allowed, as such an interval cannot be produced by the operation). Providing a minimum interval length as in the core module is not possible as this operation does not depend on the interval length, but on a calendar system. That is, one has to always generate an interval for the output nodes that matches the interval shifts plus the granularity operations. As this is a too time-intensive approach, we considered the influence on the operation in detail. As this operator only extends the length of the intervals, each smaller interval established by other operators is also part of the extended interval. As this operation has neglectable impact on the other operations, we decided to ignore the influence on the intervals of this edge during the back-propagation procedure and retrieve in the worst case always intervals that are longer than required for the program.

4.4 Existential Quantification Module

In this section, we present our extension to the core module which focuses on existential quantification for uniform weakly-acyclic DatalogMTL programs. This module is placed in the post-arity phase of the property generation, which gives us the possibility to modify the arity of the program after the full benchmark program without existential quantification has been created. In detail, for each existential variable we add to the program, we apply the following two steps:

1. We select a random node (which we call start node) and add a fresh new variable. This variable becomes existentially quantified as no source node of an incoming edge can contain this variable.

2. We forward-propagate this variable by iterating over the outgoing edges, by either adding the variable to successor nodes or pointing the variable to an existing variable in a successor node, which is not breaching the weakly-acyclic condition. We repeat this step until either we (a) reach a certain maximum path length (counting the number of edges starting from the start node), (b) have selected an existing variable, (c) or stop randomly by a user-provided factor before reaching the maximum path length.

This process ensures that we introduce only weakly acylic existential quantification. The following phases (rule composition, rule-, and data-generation) are transparent to this process and require no adaption.

Example 4.17. In the running example, the introduction of an existential quantified variable is visible at the report node where a new report ID is generated.

4.5 Summary

In this chapter, we presented a novel, open source[4], and easily extensible temporal benchmark generator. This generator allows one to generate different forms of temporal rules including recursion, aggregation, and existential quantification, is output-focused, and allows to support multiple systems. With its different phases, it is easily extensible to new operators as well as new target platforms.

We also implemented a graphical user interface, shown in Figure 4.5, to make the graph generator accessible to a wide range of users, including a save and load option to share (partial) configurations. The user-interface is split into a configuration panel for selecting parameters for the generation process on the right side, and for presenting the graph details on the left side. The left side is in addition split into a top and bottom component. The top component shows the current dependency graph (which we call graph selector) and the bottom component allows to edit the properties of a node or edge selected in the graph selector. The key components are ① Graph Generator, ② Graph Selector, ③ Rule Assignment, ④ Arity Assignment, ⑤ Property Assignment, ⑥ View and Edit of Nodes, ⑦ View and Edit of Edges, ⑧ Rule and Data Generation, ⑨ Store and Load Graph.

[4]https://github.com/kglab-tuwien/itemporal

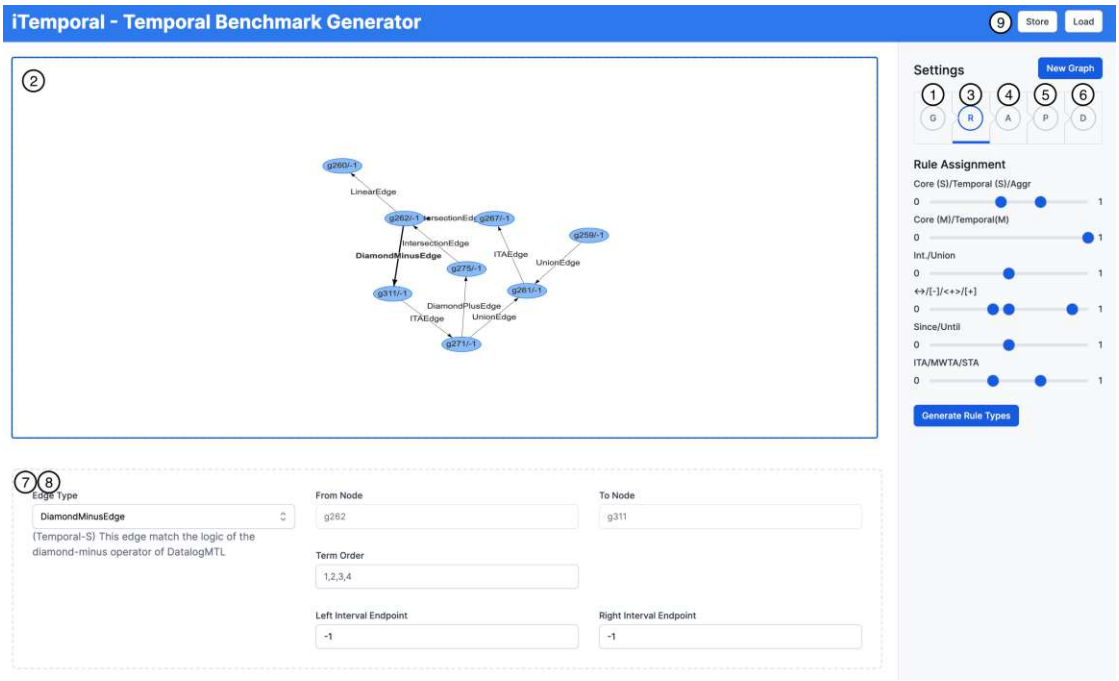


Figure 4.5: User interface of iTemporal

CHAPTER 5

System - Temporal Vadalog

With the establishment of iTemporal, an evaluation framework for measuring the system performance of DatalogMTL, it is time to establish a fully engineered system that is capable of reasoning with DatalogMTL including aggregation and existential reasoning.

In general, the development of temporal reasoners based on DatalogMTL is still in its infancy: the two implementations currently available are of experimental nature and do not satisfy these needs altogether. For example, the system proposed by Brandt et al. [BKK+17b] is implemented by the OnTop system [KBC+19]. It does not support recursion. The second system, MeTeoR [WHWCG22] supports recursive queries using a combination of materialization for non-recursive and automaton-based reasoning for checking fact entailment for recursive settings. However, despite showing efficient reasoning capabilities, MeTeoR lacks support for aggregation and basic numeric operations.

This chapter aims to make the first step towards a production-ready temporal reasoner and presents the first system that, to the best of our knowledge, supports temporal reasoning, aggregation, and negation. This chapter is mainly based on our system paper at RuleML [BBNS22][1], and includes additional experiments established during the work on DatalogMTL$^{\exists}$ as well as on the temporal benchmark generator. In Section 5.1 we discuss the functional and architectural requirements of an efficient reasoning system. Then, in Section 5.2 we illustrate the system. We evaluate the performance of different components of the system in Section 5.3. Finally, we summarize this chapter in Section 5.4.

[1] We want to note that this paper has been conducted together with another PhD Student. Section 5.1, the introduction to Section 5.2 and Section 5.2.1 are only introductory sections without technical contributions and this was conducted equally by both students. The realistic and real-world experiments (part of Section 5.3.2) and the practical parts of Section 5.2.5 have been conducted by the other PhD student, which are listed for completeness, and should not be considered as a contribution to this thesis. All other sections are personal contributions including the theoretical part of Section 5.2.5.

5.1 Requirements

Let us start the analysis of requirements again with a running example, a use case from the economic domain encoded in DatalogMTL, given in Example 5.1, which, although simplified, highlights some of the core features a production system for temporal reasoning with DatalogMTL should support.

Example 5.1. A governmental institution is supervising the changes in the corporate structure of some companies that operate in economic sectors of national strategic relevance. As part of this supervision, the institution is interested in the shareholders' actions, especially in those who are buying into the companies. For this, consider the following rules:

$$\begin{aligned} \mathit{significantOwner}(X,Y) \leftarrow \quad & \oplus_{[0,1]}\, \mathit{significantShare}(X,Y), \\ & \neg\, \ominus_{[0,1]}\, \mathit{significantShare}(X,Y) \qquad (1) \\ \mathit{watchCompany}(Z) \leftarrow \quad & \mathit{watchCompany}(Y), \mathit{significantOwner}(X,Y), \\ & \mathit{connected}(X,Z) \qquad (2) \end{aligned}$$

The atom $\mathit{watchCompany}(Y)$ denotes that a company Y is in the watchlist of the governmental institution and $\mathit{significantShare}(X,Y)$ states that entity X owns a relevant amount of shares of company Y. Rule 1 captures the dynamics of new shareholders buying in or increasing their shares. It states that if at a certain interval in the past (expressed by $\ominus_{[0,1]}$), entity X does not hold a significant amount of shares of Y, while that is the case at some point in a future interval (denoted by $\oplus_{[0,1]}$), we consider X as a $\mathit{significantOwner}(X,Y)$. Rule 2 now adds to the watchlist for every new $\mathit{significantOwner}(X,Y)$ all companies Z that are *connected* — for instance, according to the definition of connection given by other rules (omitted here) — to the new owner of shares X.

Functional Requirements. The required characteristics of DatalogMTL reasoners can be laid out along the lines of the desiderata of knowledge graph management systems [BGPS17]. They should support simple, modular, highly expressive and low-complexity fragments of DatalogMTL; they should have the ability to perform basic operations over numeric values, as well as aggregations and negation. The recent DatalogMTL fragments, such as DatalogMTL$^{\mathrm{FP}}$ [WCGKK19] and its core and linear eponymous DatalogMTL$^{\diamond}_{core}$ and DatalogMTL$^{\diamond}_{lin}$ [WCGKK20b], or DatalogMTL [WCGKK20a] over the integer timeline point in the right direction, are offering simple structure, recursion, and good complexity characteristics.

Architectural Requirements. The semantics of DatalogMTL is enforced by existing systems by inference algorithms based on time-aware variants of the well-known chase procedure [MMS79]. The native adoption of the chase presents a number of limitations in the development of production architectures. For example, it requires the entirety of the database and the generated data to be available at every possible chase step. Also,

it does not offer simple extension points, for instance to plug in different termination control policies, as discussed in Section 2.4. Specifically for temporal reasoning, this extension points are essential when infinite temporal patterns can be generated to choose between different memory management policies and select among multiple time interval merging strategies needed to handle temporal operators.

5.2 The Temporal Vadalog System

For the design and implementation of the system, we propose a novel *time-aware* reasoning architecture based on the *volcano iterator model* [GM93]. Thereby, we take inspiration from recent advances in building database and knowledge graph management systems [BSG18] and built around the core of Vadalog [BSG18], a state-of-the-art reasoner for the Datalog$^{\pm}$ family [CGP11], which we discussed in Section 2.4.

Our system offers:

- a fully engineered time-aware execution pipeline utilizing a *pipes-and-filters architecture* that enables:
  - the application of *rewriting-based optimization*, for instance, to deal with as well as to optimize more complex temporal operators;
  - *fragment aware termination strategies* which provide the ability to determine the specific fragment of DatalogMTL used in the input program to guarantee termination of the reasoning process by choosing from a range of pluggable algorithms one that exploit the specific theoretical underpinnings of the fragments;
  - a clear *interface* between temporal and non-temporal reasoning;
- *high expressive power*, implementing:
  - the DatalogMTL temporal operators natively, as well as
  - recursion and stratified negation,
  - existential quantification for uniform weakly-acyclic programs, and
  - *aggregate functions and numeric operations* over time.

In the following, we provide a thematic walk-through of the architectural components in the system, with a focus on addressing the temporal reasoning challenges, starting with a discussion of our time-aware execution pipeline.

5.2.1 A Time-aware Execution Pipeline

Along the lines of the *pipes-and-filters* architectural style [BHS07], a DatalogMTL program Π is compiled into an execution pipeline that reads the data from the input sources, applies the needed transformations, such as relational algebra operators (e.g., projection,

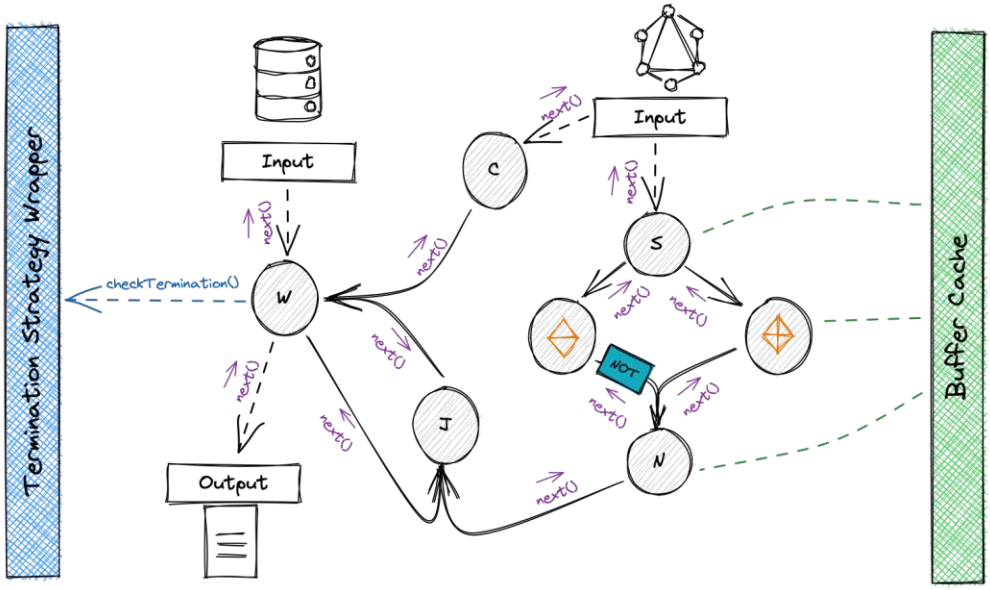


Figure 5.1: The reasoning pipeline for Example 5.1.

selection) or time-based ones, and finally produces the desired output as a result. In detail, the pipeline is built in four steps:

1. A *logic optimizer* performs a set of rewriting tasks, with the aim of reducing programs to a canonical form, where individual temporal operators are grouped together.
2. A *logic compiler* then transforms the DatalogMTL rules into a graph structure, where each component is aware of which transformation it needs to perform.
3. A *heuristic optimizer* intervenes at this point and produces variants of the generated pipeline to target higher performance, with ad-hoc simplifications.
4. A *query compiler* finally translates this logical graph structure into a *reasoning query plan*, where a *filter* is generated out of each component that will apply the transformations, and a pipe is induced by each read-write dependency between the rules.

Example 5.2. The pipeline for our running Example 5.1 is shown in Figure 5.1. The atom *significantShare* is denoted by the filter `S`, *significantOwner* by `N`, *watchCompany* by `W`, *connected* by `C`, and `J` is an artificial filter to decompose, for simplicity, the ternary join of Rule 2 into binary joins.

Runtime Model. The reasoning process then consists of a *pull-based* approach, where some rule heads are marked as *sinks* which iteratively pull data by issuing `next()` and

`get()` messages to their preceding filters. These filters in turn propagate such messages to their predecessors and eventually to a set of *source* filters that directly read from initial data sources thanks to dedicated *record managers*, i.e., data adapters. Each filter applies specific transformations, depending on the form of the associated rules (e.g., linear, joins, temporal operators, aggregations, etc.). Clearly, the `next()` primitive succeeds as long as facts are available in the cascade of invoked filters.

Example 5.3. Consider again Figure 5.1. The facts for our output filter `W` are generated directly from the input data, but also recursively from `J`.

Temporal Challenges. While the implementation and optimization of the different relational algebra operators is certainly interesting, it is not of central relevance in this thesis, where we focus on the many time-related challenges that arise and for which the Temporal Vadalog System provides support, which we list next:

- *Application of Temporal Operators.* We discuss the encoding of the temporal operators in the pipeline (e.g., the $\diamondsuit$ operator in Figure 5.1) in Section 5.2.2.
- *Merging strategies.* Facts that share the same terms and have adjacent or overlapping intervals can be simplified by merging them to a fact with a single interval and is even necessary when we want to apply the semantics of the $\boxminus$ operator correctly. In Section 5.2.3 we discuss the adoption of different merging strategies of adjacent or overlapping intervals.
- *Temporal Joins and Stratified Negation.* Temporal reasoning needs a time-aware version of the usual join (e.g., filter `W` in the figure), where the different intervals are considered when matching facts. We present our implementation, which also supports stratified negation, in Section 5.2.4.
- *Termination Strategy.* DatalogMTL allows to formulate programs where the least fixpoint operation does not terminate [BNS21b]. That is intuitively the case, as one can formulate rules that capture infinitely repeating domain events, like the repetition of weekdays. We describe in Section 5.2.5 our approach to handle termination in such scenarios.
- *Existential Quantification.* As shown in Section 3.2 the number of decidable existential fragments of DatalogMTL is limited. Our system is, to best of our knowledge, the first and at the moment only DatalogMTL reasoner that supports existential quantification for uniform weakly-acyclic DatalogMTL$^{\exists}$ programs. Furthermore, our system provides support for converting facts between DatalogMTL$^{\exists}$ and plain Datalog, to execute more advanced existential quantification tasks under well-established decidable Datalog fragments, as detailed in Section 5.2.6.
- *Aggregate functions and numeric operations.* Our system offers standard scalar and temporal arithmetic operations. Aggregate functions are also supported in the form of time-point or cross-time *monotonic aggregations*, which allows for a non-blocking

implementation that also works with recursion. To the best of our knowledge, the Temporal Vadalog System is the only DatalogMTL reasoner that implements aggregation. We have already introduced the syntax and semantics in Section 3.1.

5.2.2 Temporal Operators in the Execution Pipeline

DatalogMTL provides six temporal operators, which are pairwise symmetric; we can therefore concentrate only on the forward propagating ones: $\diamondminus$, $\mathcal{S}$, and $\boxminus$. The main idea of the reasoning pipeline is to introduce a filter node for each occurrence of an operator in a rule and feed it with the output of the operand atom. Then, the output of the operator filter is provided as an intermediate result. This process is straightforward for the $\diamondminus$ operator, which is converted into a single filter that applies a transformation of the interval according to its semantics. For the $\boxminus$ operator we require in cases of unmerged adjacent and overlapping intervals an additional pipeline filter before the temporal operator to preliminarily merge such intervals to ensure this operator captures the intended semantics (we will discuss the merging operation in Section 5.2.3). For the $\mathcal{S}$ operator, we apply a mixture of rewriting (applying the closing operator) and an extension of the join to take the required interval constraints into account.

Example 5.4. Given a dataset $\mathcal{D}$ containing the following facts

$$a@(3,4] \qquad a@(5,10] \qquad b@(2,4] \qquad b@(5,9]$$

and a program Π containing the following rules:

$$c \leftarrow \diamondminus_{[0,1]} b \qquad d \leftarrow \boxminus_{[1,4]} b \qquad e \leftarrow a \mathcal{S}_{[3,5]} b$$

then the reasoning engine will derive the following facts:

$$c@(2,10] \qquad d@(9,10] \qquad e@(8,10]$$

Optimization. In addition, we apply an optimization to chains of temporal operators by rewriting them to a single efficient temporal operator. According to the semantics of DatalogMTL, we have for a fact with an interval $\langle t_1, t_2 \rangle$ the following matching arithmetic expressions when applying a temporal operator with an interval $\langle o_1, o_2 \rangle$:

$$\langle t_1 + o_1, t_2 + o_2 \rangle \qquad (\diamondminus)$$
$$\langle t_1 + o_2, t_2 + o_1 \rangle \qquad (\boxminus)$$
$$\langle t_1 - o_2, t_2 - o_1 \rangle \qquad (\boxplus)$$
$$\langle t_1 - o_1, t_2 - o_2 \rangle \qquad (\diamondplus)$$

Note that the application of $\diamondminus$ and $\boxplus$ will produce an invalid interval (i.e., an interval, where the lower endpoint is higher than the upper endpoint) only in case both, the interval of the input and the operator are punctual and at least open at one endpoint,

while for $\boxminus$ and $\boxplus$ the subtraction easily causes invalid intervals. By definition of an interval, we have that $o_1 \leq o_2$. That is, for the combination, we add consecutive temporal operators as long as the left endpoint is smaller or equal[2] to the right endpoint and then replace all matching rules with an interval transformation of the combined interval. Similarly, to the endpoints, also the interval boundaries can be computed. That is, since the interval boundaries of the temporal operators are fixed by the rule, one only has to derive once the resulting interval boundary for an open and closed boundary of a fact by chaining together the boundary rules.

Example 5.5. Consider the following example, where (1) can be reduced to an interval $[3, 3]$ while (2) cannot, although they contain the same temporal operators, just in a different order.

$$\boxminus_{[0,5]} \diamondminus_{[0,3]} \diamondplus_{[0,2]} A \tag{1}$$
$$\diamondminus_{[0,3]} \boxminus_{[0,5]} \diamondplus_{[0,2]} A \tag{2}$$

Regarding the interval boundaries in (1): An open (closed) left endpoint results in an open (closed) left endpoint, and an open (closed) right endpoint results in an open (closed) right endpoint. Note that the resulting facts $\mathcal{D}^2$ of (2) are always a subset of the resulting facts $\mathcal{D}^1$ of (1), i.e., $\mathcal{D}^2 \subseteq \mathcal{D}^1$.

5.2.3 Merging Strategies

As discussed in the previous section, the evaluation of the box operator, that is, deciding whether $\mathfrak{M}, t \models \boxminus_\varrho A$, requires to check that for all s such that $t - s \in \varrho$, it holds that $\mathfrak{M}, s \models A$, requires the merging of adjacent and overlapping intervals, as Example 5.6 exemplifies.

Example 5.6. Given a dataset $\mathcal{D}$ containing the following facts:

$$\mathit{significantShare}(A, B)@[1.6, 1.9]$$
$$\mathit{significantShare}(A, B)@(1.8, 3.7]$$
$$\mathit{significantShare}(A, B)@(2.9, 4.0]$$

and a program Π containing the following rule:

$$\mathit{longTimeInvestor}(X, Y) \leftarrow \boxminus_{[0,2]} \mathit{significantShare}(X, Y) \tag{1}$$

where Rule 1 expresses that X is a *longTimeInvestor* of Y if X has continuously held a *significantShare* of Y for at least two years.

[2]For equal endpoints it is required that the computed interval boundaries are closed for facts with closed boundaries.

Observe that the facts of the predicate *significantShare* in $\mathcal{D}$ do not individually cover a 2-years interval, while when considered together, their combined intervals result in the fact $significantShare(A, B)@[1.6, 4.0]$. Hence, the program is able to derive the final fact $longTimeInvestor(A, B)@[3.6, 4.0]$.

When we design a software component in a reasoning pipeline, we have to make a trade-off and balance between a more streaming-based approach with the goal of staying responsive (usually coming along with a low-memory footprint and in-memory computation), and a blocking-based approach with the goal of an overall performance optimization (usually coming along with large memory occupation and a materialization of intermediate results).

This is well-known in relational systems where such a balance depends on both the semantics of the individual relational algebra operators and the optimization choices. Some operators are inherently streaming-oriented, or stateless (e.g., selection or projection), whereas others are partially or fully blocking, or stateful (e.g., join or sort) [Sci20]. Moreover, the optimizer may interleave intermediate materialization filters into the pipeline to pre-compute and store parts of it to maximize the reuse of intermediate results.

When it comes to the architecture of a modern temporal reasoning system, we recognize similar challenges when we implement the box operator, which plays the same role as data materialization in relational systems. Like the join, the box operator is partially blocking. That is, when invoked via a `next()` call, it is able to answer positively only once it has collected enough facts that cover an interval of a certain length such that the transformation implied by semantics of the box operator produces a valid interval. Yet, unlike the sort operator, once it starts to produce output, not necessarily it is finished with consuming its input facts and may produce further facts, extending the period of the valid intervals.

In the Temporal Vadalog System we offer two orthogonal options: two implementations of the box operators and three interleaving strategies.

Streaming and Blocking Box. The *streaming box* generates facts as soon as it has merged enough input facts, while the *blocking box* pulls and merges intervals until `next()` returns false and then, for each `next()` call, it forwards a single stored fact without calling the parent streams. The streaming box supports reactivity as merging is done on-the-fly and the intermediate merging results are forwarded without waiting for all the incoming data to be processed, while the blocking box reduces the amount of facts and intervals in the system.

Algorithms 5.1 and 5.2 present the logic for the blocking and the streaming strategy, respectively. Both inherit the logic of the linear filter of Vadalog to retrieve the next entry, which is visualized by a call to *super.next*(). The `next()` function returns a Boolean value denoting whether a new fact has been derived. The access to the terms is handled in successive calls to getter functions returning the value of a term's position only if required by the next pipeline step. The call *createMergeStructure* creates a data

Algorithm 5.1: Blocking strategy

```
1  mergeStructure ← createMergeStructure();
2  counter ← 0;
3  Function Next():
4      changed ← false;
5      if counter ≥ mergeStructure.length then
6          while super.next() do
7              (changed, mergedEntry) ← mergeStructure.add(getCurrentEntry());
8      if changed then
9          counter ← 0;
10     else
11         counter ← counter + 1;
12     return counter < mergeStructure.length;
```

Algorithm 5.2: Streaming strategy

```
1  mergeStructure ← createMergeStructure();
2  Function Next():
3      next ← super.next();
4      if next then
5          (changed, mergedEntry) ← mergeStructure.add(getCurrentEntry());
6          setCurrentEntry(mergedEntry);
7      return next;
```

structure for merging[3], the call of *getCurrentEntry* returns the current entry retrieved with the *super.next*() call, and *setCurrentEntry* updates the entry with the merged intervals. To exemplify the difference between strategies, let us look at a variation of Example 5.6.

Example 5.7. Consider Example 5.6 again. The *Streaming* strategy would read the first two entries, which is sufficient to apply the $\boxminus_{[0,2]}$ operator to derive the intermediate result *longTimeInvestor(A,B)@[3.6,3.7]* first, and then derives the final result *longTimeInvestor(A,B)@[3.6,4.0]*. The *Blocking* strategy would wait for all *significantShare* facts to be read first, and applies the box operator only then, thus returning only the final result.

Interleaving Strategies. The planner is equipped with multiple options to decide how to interleave explicit interval merge operations in the pipeline to achieve different performance goals or even just to guarantee correctness. We support the following options

[3]Our data structure is built around a HashMap, whose key is the fact and whose values are a collection of intervals. Currently, we use a tree-like structure as a collection that auto-merges adjacent intervals on insert.

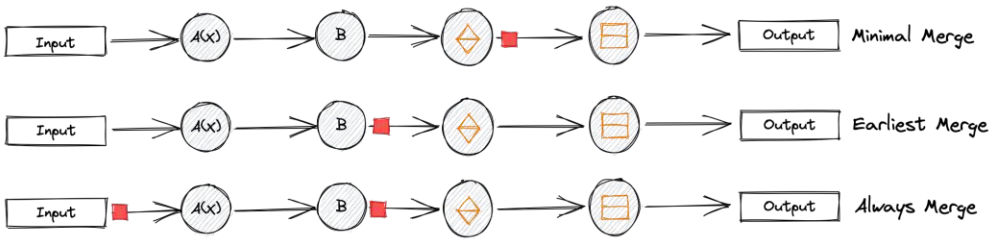


Figure 5.2: Overview of interleaving strategies

for which we provide a visualization in Figure 5.2 where merging positions are marked in red.

- *Minimal Merge.* The planner inserts a merge operation only before each box operator. In addition, one can provide hints to the planner regarding merging, such that it inserts an additional merge transformation prior to a linear transformation. Typically, such hints are useful after unions of different rules, after a diamond operator that could produce many overlapping intervals that should be combined, after the input or before the output to eliminate duplicates before showing the results.
- *Always Merge.* The planner inserts merge operations whenever the intervals are not merged. The application of temporal operators on a set of intervals will always result in a merged set, as the coalescing is applied directly on the set.
- *Earliest Merge.* If no merge operation is required (i.e., there is no box operator) the planner avoids merging; otherwise, it inserts the merge in the earliest position so that each fact contains all intervals when the box operator is reached. Operations between the merge and the box operator benefit from the reduced number of facts.

5.2.4 Temporal Joins and Stratified Negation

In Temporal Vadalog we extend the slot machine join algorithm of Vadalog [BBGS20]. This algorithm is based on an index nested loop join [GUW09], enhanced with dynamic in-memory indexing. In comparison to an index-nested loop join, there is no persistent pre-calculated index, but the index is built in-memory during the first full scan of each predicate A_k with $0 \leq k < n$ to be joined. We provide in Algorithm 5.3 a simplified algorithm that performs the temporal join only between two predicates, i.e., $n = 2$. That is, for each A_k, we first (Line 7) check whether a matching fact can be found in the index matching the known terms from the previous A_j with $0 \leq j < k$, and if not, we continue with the full scan until a fact matches in case the index is not yet fully built (Line 8-11). In case, no atom has matched (Line 12-19) we either return true in case of a join with a negated atom, continue with the next fact, or if there is no next fact, stop the process. In case an atom matches, we have to compare the intervals, as the index was designed to be unaware of intervals. This is, when a fact matches, we either intersect (join, Line

22-23) or subtract (in case of stratified negation; Line 20-21) the interval from the current partially computed interval (i.e., the computed interval up to A_j), and continue with a similar decision process as for the non-matching case (Line 24-29).

Algorithm 5.3: Temporal join between two predicates

```
   Input: predicates A_0 and A_1 to be joined
 1 I_i ← A_i.iterator();
 2 A_0(X) ← I_0.next();
 3 Function Next():
 4 |  interval ← ϱ_0;
 5 |  while true do
 6 |  |  A_1(Y) ← A_1.getNext(X);
 7 |  |  if A_1(Y) is null then
   |  |  |  // Continue full scan if index miss
 8 |  |  |  while A_1(Y) ← A_1.next(X) do
 9 |  |  |  |  A_1.addIndex(A_1(Y));
10 |  |  |  |  if Y == X then
11 |  |  |  |  |  break;
12 |  |  if A_1(Y) is null then
   |  |  |  // No further matching fact A_1 found
13 |  |  |  if A_1 is negated then
14 |  |  |  |  return true;
15 |  |  |  if I_0.hasNext() is false then
16 |  |  |  |  return false;
   |  |  |  // Repeat loop for next A_0
17 |  |  |  A_0(X) ← I_0.next() ;
18 |  |  |  interval ← ϱ_0;
19 |  |  |  continue;
20 |  |  if A_1 is negated then
21 |  |  |  interval ← interval − (interval ∩ ϱ_1);
22 |  |  else
23 |  |  |  interval ← interval ∩ ϱ_1;
24 |  |  if interval is empty then
   |  |  |  // Repeat loop for next A_0
25 |  |  |  A_0(X) ← I_0.next() ;
26 |  |  |  interval ← ϱ_0;
27 |  |  |  continue;
28 |  |  if A_1 is not negated then
29 |  |  |  return true;
```

5.2.5 Termination Strategy for the Infinite Chase of Intervals

We discussed in a related work that there exist fragments of DatalogMTL where the fixpoint computation does not terminate [BNS21b], which however admit a finite representation in DatalogMTL$^{\text{FP}}$ (and symmetrically in DatalogMTL$^{\text{BP}}$). In detail, three cases are possible for a DatalogMTL$^{\text{FP}}$ (resp. BP) program Π: (i) it is *harmless*, a sufficient condition to admit a finite model; (ii) it is *DatalogMTL$_{\diamondsuit}$ temporal linear* or *DatalogMTL$_{\boxminus}$ union free*, a sufficient condition to admit an eventually constant model under certain conditions; (iii) it is not in the previous sets but in DatalogMTL$^{\text{FP}}$, a sufficient condition to admit an eventually periodic model.

The Temporal Vadalog System guarantees termination of the reasoning process, with a two-phases *compile time* and *runtime* technique.

Compile Time. At compile time, the planner determines the fragment of Π, according to the following procedure.

- Using [BNS21b, Algorithm 1], that checks if the program has "harmful" temporal cycles, it determines whether Π is *harmless*. If so, we fix *modelKind* = *Finite*.
- Else, it determines whether Π is *temporal linear*, checking that for each rule there is at most one body predicate that is mutually temporal recursive with the head in the dependency graph of Π; if it is the case and temporal linear operators $\diamondsuit_{[t_1,t_2]}$ are such that $t_1 \neq t_2$, then we fix *modelKind* = *Constant*.
- Else, it determines whether Π is *union free*, checking that there are no rules of Π sharing the same head predicate; if it is the case and the box operators $\boxminus_{[t_1,t_2]}$ are such that $t_1 \neq t_2$, then we fix *modelKind* = *Constant*.
- Else, we are in DatalogMTL$^{\text{FP}}$ and fix *modelKind* = *Periodic*.

In all the non-terminating cases, according to [BNS21b, Lemma 2], and [BNS21b, Theorem 4], the system determines the repetition pattern length *pLength*, based on the combination of the pattern lengths of the different strongly connected components (SCCs) of Π. This will result in the production, at runtime, of facts of the form $P(\tau)@\varrho$ and $\{P(\tau)@\langle o_1, o_2\rangle, n\}$, where the intervals are given by $\langle o_1 + x * pLength, o_2 + x * pLength\rangle$ for all $x \in \mathbb{N}$, where $x \geq n$, in the periodic case, or $\langle o_1 + x * pLength, \infty\rangle$, in the constant case. All the pipeline filters are wrapped by functional components named *termination strategies*, whose goal is inhibiting the runtime generation of specific facts that may cause non-termination. All termination strategies are instructed with *modelKind* and, where applicable, *pLength*.

Runtime. At runtime, the system behaves in a fragment-aware fashion, depending on *modelKind*. If it is *Finite*, the only causes of non-termination may be the usual Datalog recursion, which is easily checked by the termination strategies with an embedded hash

index. The reasoning process will produce facts of the form $P(\tau)@\varrho$. If *modelKind* is *Constant* or *Periodic*, then the termination strategies intercept the facts generated by the "non-finite" filters and detect when they match a repeating pattern. If the model is constant, the reference interval of ground atoms is immediately converted by the termination strategies into $\langle o_1 + x * pLength, \infty \rangle$, therefore preventing the generation of redundant facts in sub-intervals; else, if periodic, the termination strategies associated to non-finite filters, generalize the numeric intervals, with their pattern-based symbolic equivalent, so that redundant sub-intervals are not generated in this case either. Example 5.8 shows one of such cases.

Example 5.8. The stock market opening days are each week from Monday to Friday. Rule 1 establishes this pattern by repeating the weekly pattern. When there is an anniversary for a company X of our multinational holding, the celebration lasts for two days (Rule 2). We want to intercept all the cases in which a celebration coincides with the stock market opening days for a company X, to study the impact on its business. In other terms, we want to compute whether *celebrationDuringOpeningDays*(A) holds (Rule 3).

$$stockMarketOpeningDays \leftarrow \diamondminus_{[7,7]} stockMarketOpeningDays \quad (1)$$

$$celebration(X) \leftarrow \diamondminus_{[0,2)} anniversary(X) \quad (2)$$

$$celebrationDuringOpeningDays(X) \leftarrow stockMarketOpeningDays, celebration(X) \quad (3)$$

The initial dataset contains the following facts, which are the initialisation of the stock market opening days as well as the list of anniversaries:

$$\mathcal{D} = \{stockMarketOpeningDays@[0, 4], anniversary(c)@[125, 125]\}$$

At compile time, the planner determines that $modelKind = Periodic$ with $pLength = 7$.

At runtime, after the generation of the fact *stockMarketOpeningDays*$@[7, 11]$, the termination strategy infers that $n = 0$, and so all facts generated by Rule 2 have the form *stockMarketOpeningDays*$@[x \times 7, x \times 7 + 4]$, for $x \geq 0$, and the generation of future intervals is stopped. It remains to apply the join of Rule 3 between *celebration*$(A)@[125, 127)$ produced by Rule 2 and the pattern generated by Rule 1. That is, computing $x \in \mathbb{N}$ such that $[x \times 7, x \times 7 + 4] \cap [125, 127)$ is not the empty interval, which holds for $x = 18$ resulting in the operation $[126, 130] \cap [125, 127)$. Thus we derive the fact *celebrationDuringOpeningDays*$(A)@[126, 127)$.

5.2.6 Combining Temporal and Non-Temporal Reasoning

We have shown in Section 3.2 that the combination of DatalogMTL with existential quantification provides only decidability for a limited number of fragments. While we support reasoning with existential quantification according to Algorithm 3.4 by introducing Skolem terms, we also provide support for more advanced existential quantification

by considering the two fragments orthogonally —in Datalog with existential quantification [BBGS20] we forbid temporal operators and in DatalogMTL we forbid other forms of existential quantification — to avoid undecidability of the program.

In order to support both modes within one program we add support for temporal wrapping and unwrapping of rules, shortly mentioned in Section 3.1 in the context of span temporal aggregation, which are of form:

$$P_1(\mathbf{s})@\mathit{temporalAtom}(\mathrm{LB}, \varrho^-, \varrho^+, \mathrm{RB}) \rightarrow P_0(\mathbf{s}, \mathrm{LB}, \varrho^-, \varrho^+, \mathrm{RB})$$
$$P_0(\mathbf{s}, \mathrm{LB}, \varrho^-, \varrho^+, \mathrm{RB}) \rightarrow P_1(\mathbf{s})@\mathit{temporalAtom}(\mathrm{LB}, \varrho^-, \varrho^+, \mathrm{RB})$$

where LB (RB) denotes if the left (right) bracket is closed and *temporalAtom* is an atom annotation to denote wrapping/unwrapping of intervals. Note that not all terms in the tuple are required in the non-temporal atom and constants can be used.

Example 5.9. Consider the conversion of a temporal fact $P_1(a)@[0, 4)$ and a non-temporal fact $P_0(b, \mathit{false}, 0, 3, \mathit{true})$ according to the provided conversion rules. These rules will map the temporal fact to $P_0(a, \mathit{true}, 0, 4, \mathit{false})$ and the non-temporal fact to $P_1(b)@(0, 3]$.

5.3 Evaluation

The evaluation section combines the results from three papers, the benchmark generator [BNS22], where we did an early evaluation of our system, the system paper itself [BBNS22], and the existential quantification paper [LNSW23], all with a different target. The benchmark paper (Chapter 4) aims to establish a baseline by comparing the system itself as well as comparing temporal aggregation with well-established time-series databases, where the target was to create a flexible and comprehensive *benchmark generator*, and provide contribution in this direction, rather than providing a basic benchmark and evaluate a large number of systems with it. The system paper (this chapter) aims to compare our system against the state-of-the-art reasoner MeTeoR for DatalogMTL, which was not available at the time of writing the benchmark paper and show the efficiency of our system. Finally, the existential quantification paper (Section 3.2) aims to measure the overhead of the introduced existential variables.

5.3.1 Establishing a baseline

As already mentioned, except our system introduced in this section, there are currently no systems that are optimized for the set of features our benchmark generator can construct. The goal of the benchmark generator is to enable such development bringing together the features we need in modern data engineering (i.e., the success of time-series DBs have shown the desire for more complex temporal reasoning, graph-based data engineering has shown the need of recursion, and data engineering in financial systems has shown the need of arithmetic and aggregation). That is the reason why we have presented in

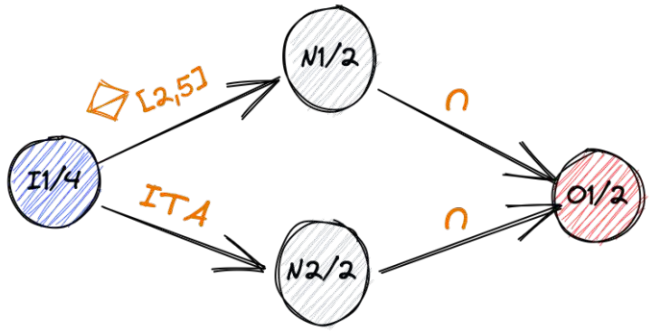


Figure 5.3: Diamond benchmark

this chapter our implementation in our reasoning system Vadalog, which is the first of its kind[4] which supports the advanced reasoning capabilities.

In this section we evaluate the implementation on a standard personal platform (MacBook Pro 13″ with M1 CPU and 16GB RAM) to provide a baseline for further systems. We believe that having established a baseline leads to competition and improved results in future systems. We first describe the chosen benchmarks generated with our benchmark generator and then report the results.

Benchmark Queries

In total, we consider three archetypal benchmark queries. These queries have been selected with the goal of including and executing different rule types of the generator to demonstrate the flexibility of the generator for generating different types of benchmark settings, as well as to mimic scenarios (shapes of graphs) encountered by various stakeholders. This includes over the three benchmark queries the application of temporal operators, (time-based) aggregations, and recursion.

Diamond Benchmark. The first benchmark is diamond-shaped and visualized in Figure 5.3. For this, we provide to the generator a predefined graph which yields in total three edge types $A = (I_1, N_1)$, $B = (I_1, N_2)$ and $C = (N_1, O_1), (N_2, O_1)$, where A and B are single-edge rules and C is a multi-edge rule. We pre-configured A with diamond ($\diamond$), B with an ITA and C with an intersection edge ($\cap$) and left the remaining decisions to the generator. The generated arity is given in the figure. We produced two variants of this program: one in DatalogMTL and one in Datalog with aggregation. This is to compare time-aware and non-time-aware reasoners.

Aggregation Benchmark. Different kind of aggregations are supported by a number of time-series benchmarks. This benchmark generates an aggregation dataset exploring

[4]Note that there exist non-optimized systems that allow to express such queries under small assumptions and with substantial rewritings, e.g., Datalog reasoners with support of arithmetic and aggregation as we show in the first benchmark query where we consider a rewriting to such a system.

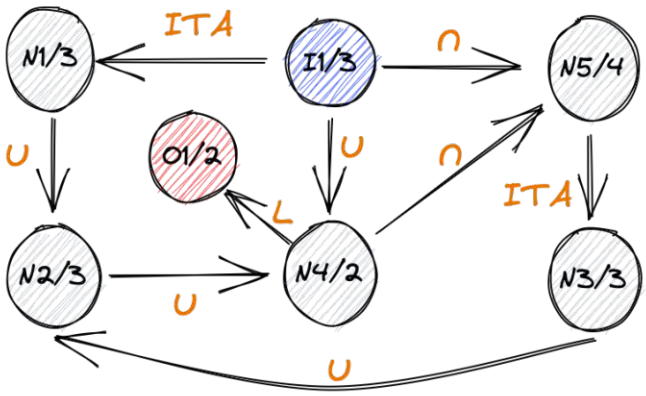
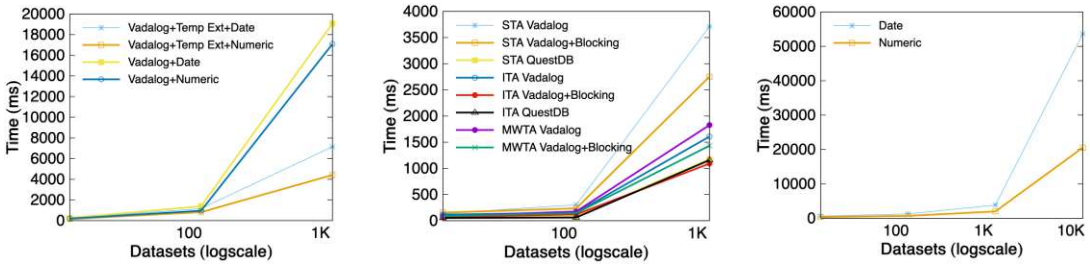


Figure 5.4: Recursive benchmark



(a) Diamond benchmark (b) Aggregation benchmark (c) Recursive benchmark

Figure 5.5: Benchmark results

the discussed aggregation types. That is, a graph with two nodes and a single edge, encoding the aggregation. For STA, we choose an interval size of a month, for window aggregation of one day[5]. This benchmark allows us to provide a comparison between our implementation and the performance of an optimized time-series database.

Recursive Benchmark. The third benchmark we consider for evaluation is a recursive benchmark. Here we create a graph that models a simulated behavior of temporal company control (visualized in Figure 5.4, where L denotes a linear edge, $\cup$ union, $\cap$ intersection and *ITA* temporal aggregation). This benchmark does not include temporal operators as it only uses temporal facts, recursion, and aggregation.

Performance Report

In the following, we report the performance of the three benchmark queries. For each query, we reused the exact program and created instances with 10, 100 and 1000 seeding values for the output nodes to get results on the performance regarding the data size.

[5]Moving Window Aggregation is not supported in QuestDB yet.

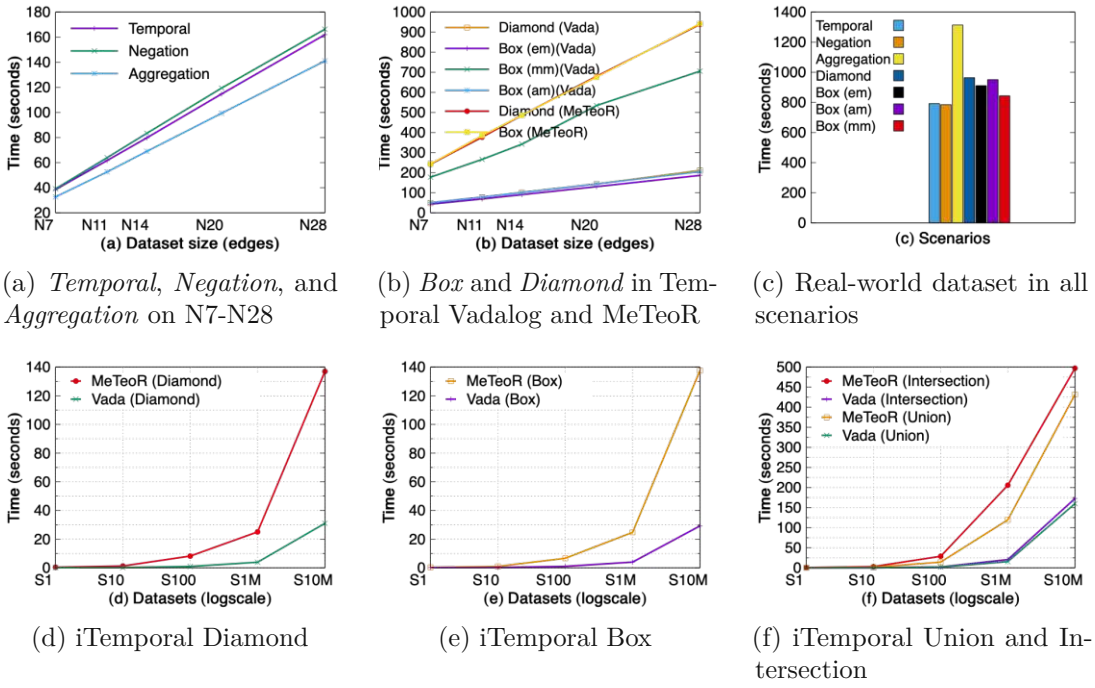
We executed each benchmark three times, each after warm-up, and reported the average execution time. We provide the results in Figure 5.5, where we report vertically the times in ms, horizontally the number of seeding values. *TempExt* stands for temporal extension, *Date* for datetime, and *Numeric* for numerical values.

Diamond Benchmark. We executed the benchmark in four different configurations, i.e., in Vadalog extended by temporal operators and Vadalog without this extension, each in two different modes. The modes distinguish how temporal data is provided to the system. We either support datetime (i.e., Gregorian Calendar) or numeric values. The results show, as could be anticipated, that date conversions have an overhead and hence one should favor simple numeric values whenever possible. The second difference is given by the temporal reasoning extension. The main advantage of the extension in this example is due to the intersection, as it exploits an optimized strategy to derive overlapping parts of intervals.

Aggregation Benchmark. The goal of this benchmark is to show the current gap of our implementation to highly optimized time-series databases for different kind of aggregations. For a fair comparison, we compare the total time of loading and querying the data and interface with both platforms over REST APIs. We use the date mode for our implementation as this is required for the calculation of the STA and restrict the generated time intervals to punctual intervals (as required by the time-series database). As already discussed, the date parsing in our implementation is quite inefficient, which is also visible in the result. An additional overhead is caused by the monotonic aggregation implementation (a fundamental feature for recursive aggregates) which forwards the intermediary aggregation result immediately to the output (i.e., it is implemented in a more streaming oriented way). To reduce the overhead, we also added a blocking aggregate operation, but there is still enough room for improvement to catch up with specialized time-series databases for such operations for STA (with an average overhead of around 250%) while we already achieve satisfactory results for ITA. Note, however, that our implementation supports aggregation over interval ranges and not only single timepoints which enables a wider range of scenarios.

Recursive Benchmark. The goal of this benchmark is to provide evidence for a real-world like scenario. Again, we execute the program in the numeric and date mode and report the values. This time, we provide an additional seed of 10000. This figure (logarithmic scale) highlights that the execution in date mode takes around twice as long as in numeric mode, but both modes scale the same way with increasing data size.

Note that we typically use the generator based on real-world data or data provided by our company graph generator (synthetic with real-world statistical properties such as skew and value distribution). This is done by providing the properties of such data in phases 1, 3 and 5 and using the instance generated in phases 1-7 together with that data. Orthogonally, critical for testing, and hard to ensure for temporal systems, our provided data generator ensures that intermediate results remain non-empty to a large extent. In this benchmark we explored the latter setting using the data of the generator.



(a) *Temporal*, *Negation*, and *Aggregation* on N7-N28

(b) *Box* and *Diamond* in Temporal Vadalog and MeTeoR

(c) Real-world dataset in all scenarios

(d) iTemporal Diamond

(e) iTemporal Box

(f) iTemporal Union and Intersection

Figure 5.6: Result Overview of Experiments

To summarize, the implementation shows a promising result for numeric values. It beats our highly efficient non-temporal reasoning system by an order of four in the diamond benchmark with one thousand seeding nodes and has an overhead of only 2.5 times compared to highly optimized time-series databases (while still using the non-efficient date implementation). We expect a drop to around 150% overhead, when one adds numeric support for this operation based on the measurements of the third benchmark.

5.3.2 System Evaluation

The second evaluation targets the comparison of our system with state-of-the-art DatalogMTL reasoners as well as targets specific implementation choices. For this, we evaluated our system in a variety of scenarios with temporal operators, recursion, negation, numerical computation, and aggregate functions, on real-world, realistic, and synthetic datasets. The performance has been compared with MeTeoR, when applicable, and against several benchmarks. The execution environment for this system evaluation is a memory-optimized virtual machine with sixteen cores and 256 GB RAM on an Intel Xeon architecture.

Experiments with Realistic and Real-World Data

We used *real-world*, and *realistic* datasets. The *real-world dataset* (RW) comes from the KG of Italian companies [BBC+20] and represents the proprietary chains from the

second half of 2019 to the end of 2021, taken at 6-months snapshots with a monthly granularity, for a total of 5 timepoints and around 31M edges evolving through time. The *realistic datasets* (N7, N11, N14, N20, N28) represent the ownerships in synthetic graphs, generated as variations on RW. These graphs evolve through time, e.g., with changes in shares, new shareholders, exit of shareholders, and so on, over five timepoints, and have from 700K to 2.8M nodes and from 2.7M to 10.8M edges. We ran the experiments on realistic and real-world data on five scenarios:

1. *Temporal*: temporal operator, recursion, and constraints on variables
2. *Negation*: stratified negation and recursion
3. *Aggregation*: aggregation
4. *Diamond*: diamond operator and recursion
5. *Box*: box operator and recursion

We executed each scenario against the datasets N7-N28 and RW in the Temporal Vadalog System and, for *Box* and *Diamond*, also in MeTeoR — scenarios that include features that are supported by MeTeoR. For *Temporal*, *Negation*, and *Aggregation* we performed three runs each and averaged the elapsed time; the merging strategy was *always merge*. For the *Box* scenario, we tested *minimal*, *earliest* and *always merge*. We used an one-hour timeout to abort long-running experiments.

Figure 5.6a shows the performance of our system on the scenarios *Temporal*, *Negation*, and *Aggregation* over the realistic datasets N7-N28. We have achieved good scalability, with a linear increase in the elapsed time. The more expensive *Negation* runs at just over 166 secs for the biggest dataset, N28, while *Aggregation* performs best at 32-141 secs, given the non-recursive setting.

Figure 5.6b shows the *Diamond* and *Box* scenarios comparatively with MeTeoR. Temporal Vadalog is 80% faster than MeTeoR in the *Diamond* scenario. For *Box*, Temporal Vadalog is 80% faster with *earliest merge* (*em*) and *always merge* (*am*). This test also highlights the importance of the merging strategy choice. In fact, in the case of *earliest merge* the performance is better, ranging from 43 to 186 secs, while the *minimal merge* requires four times of it, as more data is sent through the pipeline until the merging operation is applied. In the case of *earliest merge*, the merging operations are concentrated, and the same happens for *always merge*.

Finally, Figure 5.6c shows the performance of the five scenarios (plus three merging strategy variations for *Box*) on our *real-world* dataset. While MeTeoR exceeds the established timeout in all applicable scenarios, our system always terminates within one hour, with the *Temporal* and the *Negation* scenarios being the best, having an elapsed time of 780-790 secs (∼13 mins). Comparing the rules of *Temporal* where execution takes around ∼13 minutes with *Diamond* where execution takes around ∼16 minutes, one can

explains the performance improvement for the *Temporal* scenario. While both scenarios share the same rules, the *Temporal* scenario contains one additional variable constraint, which allows to skip many edges in the graph, thus yielding less facts to be processed in the remaining program. The *Box* scenario shows differences of around 2 minutes between the strategies, favoring minimal merge due to less overlapping intervals in the graph.

Temporal Foundation Benchmark

Observing a substantial speedup of the Temporal Vadalog System with respect to MeTeoR in the realistic scenarios, we generated specific temporal benchmarks to confirm this aspect. In particular, we compared the systems as of their main temporal operations, that is, temporal operators, joins and unions. To generate our benchmarks, we used our benchmark generator *iTemporal* to generate DatalogMTL programs with different data sizes between 1K to 10M (S1-S10M) facts per input atom randomly distributed over a given domain. Figures 5.6d-f present the results. For all operations, we see that our system outperforms MeTeoR with a factor of 3 to 4 (depending on the benchmark) for $10M$ facts.

5.3.3 Performance of Existential Quantification

In this section, we present an experimental evaluation of our reasoning approach from Algorithm 3.4 for reasoning in DatalogMTL$^{\exists}$ under the uniform semantics.

All experiments were performed on a personal computing platform, namely a MacBook Pro 13″ with M1 CPU and 16GB RAM. Each experiment computes the materialization of the model, as in Algorithm 3.4, three times and the average time is reported.

The Performance Cost of Existential Rules

In the first series of experiments, we study the performance impact of adding existential rules. For this, we created four instances I1–I4 (that are pairs of DatalogMTL$^{\exists}$ programs and datasets) that vary in the existence/non-existence of recursion in the programs, the number of rules, the number of existential positions, and the size of datasets, as described in Table 5.1. Instance I1 contains a program with a single linear rule, I2 has a complex non-recursive program, I3 has a simple recursive program (with no 'recursion via time'), and I4 has a complex recursive program.

We compared the running times on instances I1–I4 and on their counterparts without existential rules; the results are reported in Table 5.2. Clearly, inference times in the DatalogMTL$^{\exists}$ cases are expected to be higher. The experiment shows that a total overhead caused by existential rules was between 5% and 35%. The increase in time is only moderate and primarily seems to correlate with the number of existential positions in a program. The impact of the size of the dataset is mostly relevant for I2, where the number of facts in the final materialization increases by 16% due to existence of existential rules. In the case of I3, the increase is around 1% while in the remaining instances the increase is neglectable.

	recursive?	# rules:	# exist. positions:	# facts:
I1	no	1	1	10M
I2	no	14	3	2M
I3	yes	4	1	20M
I4	yes	23	3	200k

Table 5.1: Overview of generated instances

	I1	I2	I3	I4
DatalogMTL	83.3 s	16.6 s	47.9 s	9.4 s
DatalogMTL$^{\exists}$	91.4 s	22.4 s	50.5 s	11.5 s

Table 5.2: Comparison of DatalogMTL and DatalogMTL$^{\exists}$

		# of exist. positions:		
		1	5	10
paths' length	1	15.4 s	16.8 s	19.6 s
	5	15.0 s	8.9 s	20.5 s
	10	17.3 s	19.5 s	11.4 s

Table 5.3: Performance for different path lengths and number of existential positions

Existential Positions and Length of Propagation

In the second experiment, we investigate in more detail the impact on the performance of the form of existential rules. For this, we generated a recursive DatalogMTL$^{\exists}$ program with 20 rules and a dataset with around 20000 temporal facts, for which reasoning takes 13.77 seconds. Next, we have extended the program by adding existential rules in several ways, which resulted in programs differing in the number of existential positions and lengths of paths in the dependency graphs which contain existential positions. Our results are summarized in Table 5.3.

Our experiment reveals that the performance depends on the interactions of existential rules with the remaining part of the program and with the dataset. Interestingly in some cases this interaction leads to a decrease of the running time, so that the running time becomes smaller than in the case of no existential rules. Indeed, one of such interaction we observed is as follows. Due to existential positions in a program, our procedure can derive facts which cannot be coalesced, and so, other rules with boxes in bodies cannot be fired. This stops derivations of more facts, which are present in the corresponding program without existential rules.

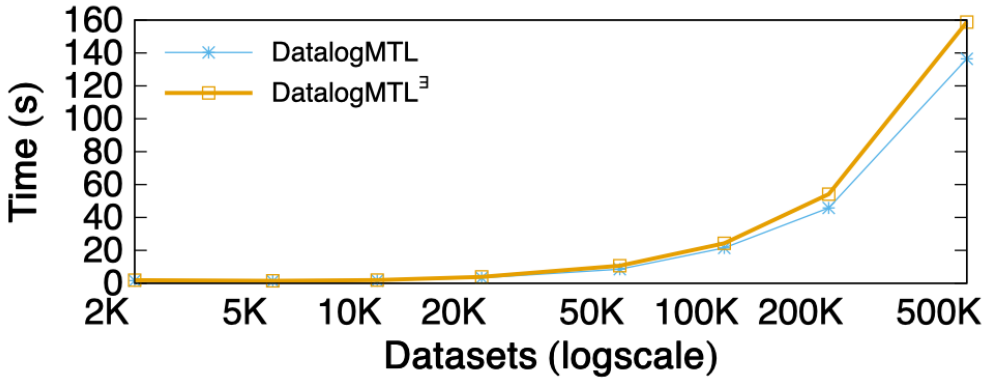


Figure 5.7: Scalability of our implementation

Scalability

Finally, we tested the scalability of our implementation. For this, we use the recursive program from the previous experiment (20 rules) with 1 existential position and path length 10, together with datasets of increasing size, namely with 2k, 5k, 10k, 20k, 50k, 100k, 200k, and 500k facts[6]. The results of this experiment are reported in Figure 5.7. We see that our implementation scales linearly for the given program and dataset and despite the high theoretical complexity of DatalogMTL$^{\exists}$, we observe inference times that are feasible for practical applications even on large instances. In particular, we observe an average running time of 159 seconds for the largest test instance, which contains 500k facts.

5.4 Summary

In this chapter, we presented a novel architecture and system for reasoning with DatalogMTL, delving into our optimization techniques, including specific termination strategies to detect *periodic models* as well as *query rewriting* methods. We emphasized its performance and scalability with various benchmarks, i.e., a mixture of fundamental benchmarks testing individual operators as well as programs provided by stakeholders, and outperformed state-of-the-art reasoners in this language. Furthermore, we study the practicality of DatalogMTL$^{\exists}$ via a first implementation. Through a series of experiments, we illustrate that despite high theoretical worst-case complexity, reasoning in DatalogMTL$^{\exists}$ can be feasible in practice.

[6]Note that the datasets differ by three orders of magnitude.

CHAPTER 6

Applications

Distributed ledger technologies (for example, blockchains) provide the foundation and core infrastructure for decentralized finance (DeFi), a type of finance that does not rely on intermediaries like exchanges, banks or brokers [Sin21]. DeFi applications are built by utilizing smart contracts, an executable code that facilitates the process of executing and enforcing the terms of an agreement between (untrusted) parties [AvM17].

Example 6.1. Consider a contract that models the following situation: A person lends from a private funding company ten Bitcoins with an interest rate of 3% per year for three years. The interest payment is monthly.

It has been shown that the usage of logic-based smart contract languages allows to better represent and reason upon the conditions of a smart contract [AvM17, IGRS16]. In particular the temporal dimension is of high interest when modeling smart contracts, as even in simple examples such as Example 6.1 it is a natural component. Yet, all approaches [IGRS16, FN16, SU19, HZ18, CMMO19, SD20] to the best of our knowledge either neglect the temporal domain at all, handle it in a sub-optimal way, or miss the support of a widely accepted secure blockchain technology.

In this chapter we study the applicability of DatalogMTL as a smart contract specification language with the aim to make it translatable to Solidity, a widely-accepted and the main programming language for Ethereum, one of the largest blockchains.

This chapter is based on our recent papers on making DatalogMTL accessible to the financial community [NS22a] and the Datalog community aware of blockchain related topics [NS22b]. We start in Section 6.1 by providing a high-level introduction to blockchains and continue by discussing the requirements of a DatalogMTL-based smart contract language in Section 6.2. Then, in Section 6.3 we discuss the main components which are required by a smart contract language and formally define a DatalogMTL-based smart contract. We discuss preliminary results on a DatalogMTL to Solidity converter in

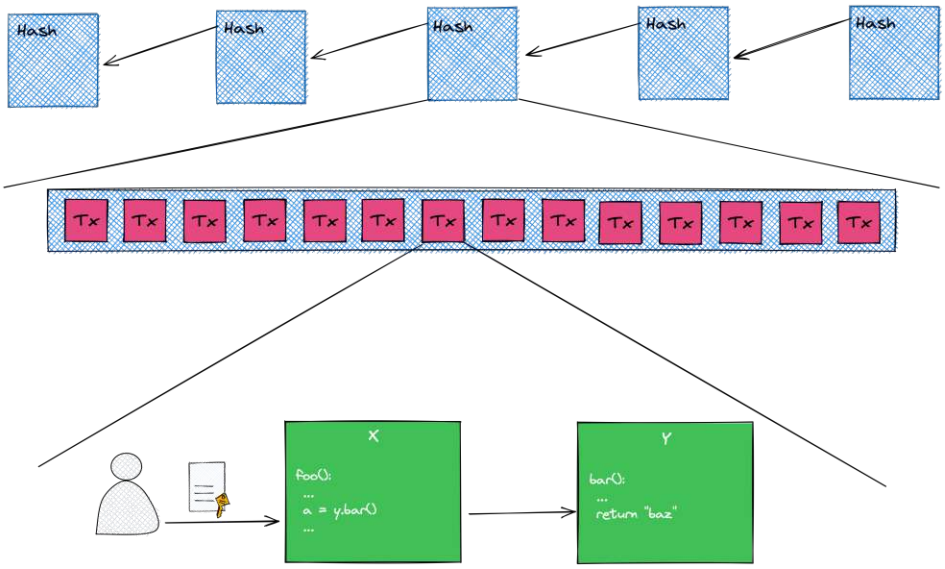


Figure 6.1: Blockchain 101

Section 6.4 and provide a case study in Section 6.5. Finally, we summarize the chapter in Section 6.6.

6.1 Blockchain 101

A distributed ledger technology (DLT) is a decentralized, immutable, and append-only database across different nodes that is managed by multiple participants. At the core of the DLT is a consensus mechanism which contains procedures and rules on how transactions are validated by the nodes [10121].

A blockchain is a form of DLT, where transactions are recorded and grouped into blocks, where each block includes a hash of the previous block. Other forms of DLTs are Tangle [Pop18], built on top of a directed acyclic graph, or Corda [HB16], a leading DLT for regulated industries where the ledger stores per node only the facts the node is aware of that they exist.

Smart Contracts are programs stored on the distributed ledger technology that run when certain conditions are satisfied. Usually, smart contracts manage an agreement (i.e., rules) between multiple parties per code without requiring a third party. They reduce risk due to the tamper-proof property of DLTs and provide transparency to the process. Typical smart contract applications include voting, supply chains, mortgage, copyright protection, or employment arrangements [Anw18].

Figure 6.1 visualizes a blockchain with a block containing multiple transactions (Tx), and a transaction that is signed by a user and invokes a method of a smart contract which interacts with another smart contract.

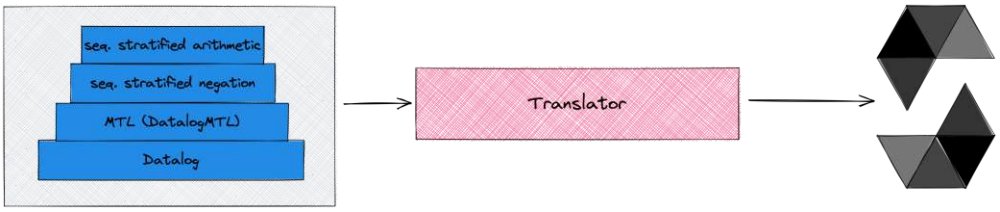


Figure 6.2: Overview of approach

6.2 Requirements

In this section, we provide what we think are the most essential temporal requirements a logic-based smart contract language has to support to provide a minimum amount of useful reasoning capabilities for DeFi applications.

1. *Validity Interval.* It is necessary to represent intervals, i.e., when a specific kind of operation is valid, and not only punctual points. For example, voting contracts specify an interval when voting is allowed.

2. *Periodicity.* It is necessary to specify periodic patterns that encode repeating agreements, for example, to encode that the salary is paid per month.

3. *One Time Event/Delay.* It is necessary to encode single future events. For example, in shopping contracts a payment reminder has to be sent in case the money has not been paid.

4. *Negation.* While only indirectly related to the temporal domain, it is necessary to support negation. This is illustrated in the third point of this list, which encodes that something has not occurred within a specific interval.

5. *Verification.* It is necessary to verify whether a given model is satisfied for a given set of rules. This helps for the verification of certain properties and is in line with the work of using temporal logic for verification of smart contracts.

Note that DatalogMTL supports intervals (1) as a core concept, periodicity (2) by recursive rules, delays (3) by temporal operators, (4) by stratified negation, and (5) by the rule head $\bot$ out of the box. Yet, for many programs one requires a more relaxed form of stratified negation, namely sequential stratified negation [Zan12]. This form allows the usage of negated body atoms in recursive rules in case the negated body atoms maintain the monotonicity property of DatalogMTL. Furthermore, analogously to sequential stratified negation, we allow sequential stratified arithmetic (i.e., $+,-,\times,\div$). Figure 6.2 provides an overview of our approach including the stack of used language features.

6.3 Language Definition

In this section we discuss the modelling of smart contracts with DatalogMTL over the integer timeline [WCGKK20a]. We start by formally defining DatalogMTL-based smart contracts and then explain the individual components in detail with a running example. Note, that smart contracts often follow best-practice patterns, such as in Solidity [Eth22], the leading smart contract language, which we explore in Example 6.2.

Definition 6.3.1. A smart contract is a quadruple $(N, \Pi, \mathcal{D}, A)$ where:

- N is a unique name of the contract (namespace).
- Π is a forward propagating DatalogMTL program encoding the rules of the smart contract.
- $\mathcal{D}$ is the initial dataset encoding the initial state of the smart contract.
- A is the set of activators containing predicates which may be invoked by other smart contracts or parties.

Example 6.2 (Running Example)**.** We consider one of the main patterns here, namely that contracts often are modelled as state machines. Let us consider a state machine with the following four stages: init, acceptingBids, execution, and finished. A contract is being initialized, stays in the "accepting bids" state for exactly 10 days, then is in execution and finally finished.

Initial Dataset. In Solidity one usually sets up the initial values with the constructor which is invoked when deploying the smart contract in the blockchain. In DatalogMTL this corresponds to providing the initial values as a given dataset at a certain timestamp. For the timestamps, we have two options:

- *Local timestamp.* The smart contract does not depend on any other timestamps (either time-information such as the current date or information from other contracts). Then one can initialize the timestamp t with zero or any other number.
- *Global timestamp.* The smart contract uses the timestamp of the blockchain for initialization. This allows to use a shared timestamp and the actual current time inside the contract. As for the contract itself, the initial value has no influence on the reasoning (it is just shifted to a different timestamp), we always use a globally shared timestamp. That is, we set the $t = now()$, where now is the current timestamp in the blockchain.

Example 6.3. (continued) The initial values are given by the following dataset encoding the initial stage of the state machine: $\{stage(\text{init})@t\}$, where t is the deployment

timestamp as discussed above.

Invoking a Smart Contract. Usually, to invoke a smart contract in Solidity one calls a method of a smart contract. Each call is initialized by a transaction, which allows the called smart contract to invoke other smart contracts. In DatalogMTL we only have rules that fire (i.e., derive the rule head) when the body is satisfied. In order to model an invocation of a method in DatalogMTL, we mark predicates as *activators*. These activators can be "called" by adding the activator for the current timestamp to the dataset. By writing rules that use these activators in the body one triggers the derivation of further facts. As we use activators for triggering rules, they have to obey the following restrictions to enforce compatibility with Ethereum: (i) an activator is only allowed in the body of the rule, (ii) only one activator is allowed to be executed per timestamp (to have an implicit method call order and prevent conflicting states).

Example 6.4. (continued) The set of activators for the bidding state machine consists of the following activators: $\{bid, execute\}$.

Program. One usually encodes in a method of a smart contract the transition of data/state before and after the execution of a method. We already covered in the previous paragraph the activation via an "activator". This activator triggers a set of rules that state how the current dataset at timepoint t is changed for the next state at timepoint $t+1$. This implies that such rules are not allowed to derive facts for entries in the past.

Example 6.5. (continued) The rules for the state machine are given as follows:

$$\boxplus_{(0d,10d)} stage(\text{acceptingBids}) \leftarrow stage(\text{init}) \quad (1)$$

$$\boxplus_{[10d,10d]} stage(\text{execution}) \leftarrow stage(\text{init}) \quad (2)$$

$$\boxplus_{[1,1]} stage(\text{execution}) \leftarrow stage(\text{execution}), \neg execute, bid(\_,\_) \quad (3)$$

$$\boxplus_{[1,1]} stage(\text{finished}) \leftarrow stage(\text{execution}), execute \quad (4)$$

$$doAction\ldots \leftarrow stage(\text{execution}), execute \quad (5)$$

$$vBid(\text{S},\text{A}) \leftarrow stage(\text{acceptingBids}), bid(\text{S},\text{A}) \quad (6)$$

$$\boxplus_{[1,1]} highBid(\text{S},\text{A}) \leftarrow vBid(\text{S},\text{A}), highBid(\_,\text{CA}), A > CA \quad (7)$$

$$\boxplus_{[1,1]} highBid(\text{S},\text{CA}) \leftarrow vBid(\_,\text{A}), highBid(\text{S},\text{CA}), A <= CA \quad (8)$$

$$\boxplus_{[1,1]} highBid(\text{S},\text{CA}) \leftarrow highBid(\text{S},\text{CA}), \neg vBid(\_,\_), bid(\_,\_) \quad (9)$$

$$\boxplus_{[1,1]} highBid(\text{S},\text{CA}) \leftarrow highBid(\text{S},\text{CA}), \neg vBid(\_,\_), execute \quad (10)$$

Rule 1 defines the duration of the acceptingBids stage, Rule 2 marks the start of the execution and Rule 3 extends this stage in case there was no *execute*-action. Rule 4 translates the stage to finished in case of an execution and Rule 5 simulates a possible

action of the execution. Rules 6-10 manage the bidding phase, which defines on how the data is changed in the possible settings (there is a bid which is higher/not higher and there is no bid). Note, that we model also the "unchanged" values (3, 9-10) in DatalogMTL to derive a well-defined next state (requiring always some given activator for a given timepoint for executing the rules only for the eligible time).

Namespaces. So far, we focused on modelling a single smart contract. However, a method of a smart contract can invoke a different (target) smart contract and use the return value in the remaining method. In order to handle this case in DatalogMTL, we require the following:

- *Namespaces* to uniquely identify a given smart contract.
- *Triggers* to "trigger" an activator of a different smart contract. Note that due to the restrictions of one activator per timestamp, this creates a temporal chain of calls.
- A *Default Activator* (Caller.default) to handle the passing of "return values" to the calling smart contract. In case different default activators are required, one can introduce additional intermediary smart contracts, which default activator triggers a named activator of the original contract.

Example 6.6. (continued) A namespace for the running example could be "Biding-Contract" and the *doAction* could be an invocation of a contract (e.g., namespace *Token*) to transfer tokens, e.g., $\mathit{Token.transferFrom}(\mathrm{Src}, \mathrm{Tgt}, 20)$. After a successful application it uses the default activator to trigger the calling smart contracts so that it can continue the execution.

Blockchain-specific Parameters While the introduced concept so far allows one to model several use cases, there is one open point for discussion, namely blockchain-specific parameters. In the following, we study the most important properties of Solidity and how we can ensure the usage in DatalogMTL. Typically, one distinguishes between three different classes of properties:

- *Block-specific* properties, such as the block number, or the timestamp.
- *Transaction-specific* properties, such as the initiator of the transaction (origin) considering the full chain of executions starting from an external call to the activator.
- *Message-specific* properties, such as the caller or the number of coins sent to the invoked contract.

We inject these properties by introducing pre-defined predicates which are added to the dataset, e.g., $\mathit{block}(\mathrm{Number})$, $\mathit{transaction}(\mathrm{Origin})$ or $\mathit{msg}(\mathrm{Caller})$. The only property we

Figure 6.3: Overview of translation process

do not consider in this mapping is the timestamp as this causes ambiguity and hence has to be treated in a special way, which the following two points highlight:

- *Timestamp per block.* In Ethereum, the timestamp is provided per block and the only requirement for the timestamp is that it is higher than the previous block. This means, that the timestamp received in a method is the timestamp of the block. That is, each transaction, and hence each method invocation (i.e., message) shares the same timestamp and the actual timestamp is dependent on the miner of the block and may not model the exact time.

- *Time and ordering.* For modelling smart contracts, we require two different kinds of timestamps. One that models the real time, for example to check if something happened within the last 24 hours, and one to model the sequential order of method invocations.

In order to handle this representation problem in DatalogMTL, we decided to assign each message its own timestamp. This is possible as the main goal is the conversion to the Ethereum platform, where the natural mapping discussed above can be used, which yields a quite natural, relatively simple structure of time.

6.4 Translation Engine

In the previous section we established the required formalism to model an Ethereum-compatible smart contract in DatalogMTL. In this section, we present our proof of concept for translating DatalogMTL rules into Solidity. Figure 6.3 shows a general overview of the process, where each step is discussed in the following. We want to remark that Solidity is Turing-complete, hence more expressive then DatalogMTL without arithmetic and thus allows to formulate more complex programs. Yet, in the examples studied with stakeholders we identified that the given examples can be formulated in a logic-based language with the integration of the discussed extensions in Section 6.2.

Phase 1 - Parser. The first phase is all about parsing and normalizing the program. For this, we read a DatalogMTL smart contract from file and apply a normalization procedure to ensure that each rule has a head without a temporal operator and the body either has literals containing no temporal operator or consists of exactly one literal with a single temporal operator.

Example 6.7. (continued) Rule (3) of the running example would be rewritten by introducing an auxiliary predicate and moving the temporal operation into the body as follows:

$$temp1(\text{execution}) \leftarrow stage(\text{execution}), \neg execute$$
$$stage(\text{execution}) \leftarrow \diamondminus_{[1,1]} temp1(\text{execution})$$

Phase 2 - Grouping of Rules and Method Detection. The goal of this step is to group rules executed together and sort them by execution order to create for each group a method in a Solidity contract. By the usage of the activators, we directly derive the starting points of these methods. These points can be used to trace the derivation to check whether additional rules are activated. Next to that, we find some other common patterns in rules:

- *Negated activators* are often used for describing the continuation of the current state which does not require any handling in Solidity. If this is not the case, such negated activator has to be checked before each method call.
- *Global rules* may not be triggered by any activator, i.e., they can depend on the current state of the smart contract. Similar to the second case of negated activators, these rules have to be enforced before any method call. Such rules are detected by checking the body of the rule whether it is not only directly enforced by a rule containing an activator, e.g., there is an initial state or some time delay in between.
- *Initial rules.* These are similar to global rules but are only executed on deployment of the smart contract. These rules typically live in the constructor and have no time-dependent condition in the future (as otherwise they would be global rules).

Example 6.8. (continued) The group of rules for the running example are the execution group (4-5) containing the *execute* activator, the bidding rules (6-8) containing the *bid* identifier and the global group (1-2) which are activator independent. Rules (3), (9) and (10) are detected as negated activators copying the current state and are removed as no further consideration in Solidity is required[a].

[a]Note that we have added those rules at the beginning to allow the smart contract to be interpreted by a Datalog reasoner. When the only goal is the conversion to Solidity, one can ignore the rules when writing the smart contract in DatalogMTL.

Phase 3 - Data Types. We distinguish between the following two types:

- *Term Type.* We derive the data type of a term where possible from the internal values (e.g., from the dataset). However, when there is an activator, we are unaware of which values are provided by the user (e.g., is it 8bit or 256bit integer). Therefore, we extend the activator syntax to provide data types for the activators.

- *Atom Type.* An atom can be either stored by one or more variables, each either being a primitive, array or map. First, we distinguish whether there exists only one valid atom per time unit, then single values are sufficient and per term a variable is created. Otherwise, we check if there is a pattern for accessing the atom to identify a map type (e.g., there is an access in body and head and some variables are shared).

Example 6.9. (continued) The activator for bidding would be *bid*(*Int*). Note that with the introduction of the message-specific parameters the sender S is extracted into an own atom *msg*(Caller). The atom type of *highBid* would consist of multiple primitives.

Phase 4 - State Types. For the state of the smart contract, we have to distinguish between *local state* where the derived fact is not shared with any other method and does not need historical information, state variables where we only require the *latest state* (are not time-dependent), variables where we need a *last-timestamp of action*, and variables where we need a *history of changes* for a time period. For detecting the local state, we use the grouping to check whether there exists no other group that contains the same atom and uses information from other time units. In such a case, we can inline the state at contract generation. For detecting the latest state atoms, we check whether the state always depends on the previous timestamp in DatalogMTL. To distinguish between the remaining cases, we differentiate on whether we have to check if some condition is currently fulfilled or whether there is a more complex comparison where the other case is not enough[1].

Example 6.10. (continued) Variables Stage0, HighestBid0 and HighestBid1 are identified as global state, where the numbers reference to the term position. In addition, a variable storing the timestamp when *stage*(init) becomes true is created.

Phase 5 - Contract Generation. What now remains is the processing of the rules per group. We start by adding preliminary checks, then continue per activator. That is, we structure the rules by a dependency graph and convert each rule to Solidity. In case there is an option to handle multiple rules at the same time, we try to find specific structures that follow a specific pattern. For example, for the bidding rules, we detect that these rules are a typical if-else structure and furthermore that only the if part is relevant.

Example 6.11. (continued) The generated function for bidding is as follows, with checks for the correct phase and a valid offer. Note that `__global` is a modifier executed before the function that handles automatic state transitions. The full program is given in Figure 6.5.

[1]Note that, by definition, non-local state is only allowed to be set for future time points.

```
contract ERC20 {
    mapping(address => int) _balanceof1;
    mapping(address => mapping(address => int)) _allowances2;

    constructor() {
        _balanceof1[msg.sender] = 2000;
    }

    function transfer(address to_, int value_) public {
        require(value_ <= _balanceof1[msg.sender], "invalid request");
        int newbalance1_ = _balanceof1[msg.sender] - value_;
        int newbalance2_ = _balanceof1[msg.sender] + value_;
        _balanceof1[msg.sender] = newbalance1_;
        _balanceof1[to_] = newbalance2_;
    }

    function approve(address spender_, int value_) public {
        _allowances2[msg.sender][spender_] = value_;
    }

    function transferFrom(address from_, address to_, int value_) public {
        require(_allowances2[from_][msg.sender] >= value_, "invalid request");
        require(value_ <= _balanceof1[from_], "invalid request");
        int newbalance1_ = _balanceof1[from_] - value_;
        int newbalance2_ = _balanceof1[from_] + value_;
        _balanceof1[from_] = newbalance1_;
        _balanceof1[to_] = newbalance2_;
        int newvalue3_ = _allowances2[from_][msg.sender] - value_;
        _allowances2[from_][msg.sender] = newvalue3_;
    }
}
```

Figure 6.4: Generated ERC20 Solidity contract

```
function bid(int a_) public __global {
  require(compareStrings(_stage0, "acceptingBids"), "invalid request");
  require(a_ > _highestbid1, "invalid request");
  _highestbid0 = msg.sender;
  _highestbid1 = a_;
}
```

6.5 Case Study

In this section, we discuss the results of our smart contract language and generation process by encoding the running example as well as the ERC-20 standard for tokens in our language and comparing the generated code with the original Solidity code [Eth18, Eth22]. The results of our generation are given in Figure 6.4 as well as Figure 6.5.

In general, the generation of the smart contracts shows similar code as the compared original code. This means, that the supposed steps of the pipeline achieve the desired goals. In a detailed comparison, we detected some possible improvement potentials: (i) code reduction, (ii) check for variable range (e.g., overflows), and (iii) improvement of variable types to reduce execution costs.

Besides the discussed limitations, we want to remark that the current implementation is a proof-of-concept not supporting all cases yet, especially it misses support of handling looping structures. Furthermore, we identified that the detection of the required state types and their dependencies given by the rules requires more detailed investigation. Still, as the results have shown, the generation process is feasible, and we continue sharpening the generator to cover more advanced tasks in the future.

6.6 Summary

In this section, to the best of our knowledge, we introduced the first temporal logic-based smart contract language that is executable on a widely used blockchain. While it is still a proof of concept, it already handles many of the subtleties and challenges, in particular blockchain-specific variables, or state types. Our translator is able to produce meaningful code for widely used patterns, including the ERC-20 token and the state machine.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.7;

contract BIDINGCONTRACT {

    string _stage0 = "InitState";
    address _highestbid0;
    int _highestbid1;
    uint stageInitStateUpdated = block.timestamp;
    uint stageInitStateUpdatedLastFired;

    constructor() {}

    modifier __global() {
        if(stageInitStateUpdatedLastFired < stageInitStateUpdated &&
           block.timestamp > stageInitStateUpdated + 0.0 days &&
           block.timestamp < stageInitStateUpdated + 10.0 days) {
            stageInitStateUpdatedLastFired = stageInitStateUpdated;
            _stage0 = "AcceptingBids";
        }
        if(stageInitStateUpdatedLastFired < stageInitStateUpdated &&
           block.timestamp >= stageInitStateUpdated + 10.0 days) {
            stageInitStateUpdatedLastFired = stageInitStateUpdated;
            _stage0 = "ExecutionState";
        }
        _;
    }

    function bid(int a_) public __global {
        require(compareStrings(_stage0, "AcceptingBids"), "invalid condition");
        require(a_ > _highestbid1, "invalid request");
        _highestbid0 = msg.sender;
        _highestbid1 = a_;
    }

    function execute() public __global {
        require(compareStrings(_stage0, "ExecutionState"), "invalid condition");
        _stage0 = "Finished";
    }

    function compareStrings(string memory a, string memory b) public pure returns (bool) {
        return keccak256(abi.encodePacked(a)) == keccak256(abi.encodePacked(b));
    }
}
```

Figure 6.5: Generated running example Solidity contract

CHAPTER 7

Conclusion

In this thesis we enabled temporal reasoning with DatalogMTL for knowledge graphs by advancing the state-of-the-art in several directions: by supporting new features, namely aggregation and existential quantification, by creating a benchmark generator as an evaluation tool, by developing a fully engineered reasoning system and by studying new applications for DatalogMTL.

In this chapter, we examine in Section 7.1 the research questions in retrospect and highlight the key findings. Then, in Section 7.2, we look ahead and explore open points that should be addressed in future work.

7.1 Summary

In this section, we summarize and discuss the answers to the research questions introduced in Chapter 1. For each main chapter of the thesis, we first recall the context of the chapter, and then restate the research question from the introduction with a brief discussion about the most significant results.

The first two research questions addressed the missing *functionality* for temporal reasoning in knowledge graphs with DatalogMTL by adding two key capabilities: aggregation and existential quantification.

Research Question 1. In which form can monotonic aggregation, as used in Datalog, be used over temporal data, i.e., in DatalogMTL?

> In Section 3.1 we introduced different kind of time-point aggregations for DatalogMTL. We showed that it is possible to reduce time-point aggregation to the classical aggregation of Datalog, with a coalescing of adjacent intervals. Furthermore, we introduced a granularity operator that is capable of extending intervals to the granularity of weeks,

months, or years and thus is able to provide the right interval boundaries for the span temporal aggregation. Finally, we discussed time-axis aggregation to summarize the behavior of values over the timeline, giving a concrete semantics for monotonic increasing and decreasing intervals.

Research Question 2. What is the complexity for different restrictions of existential quantification? Is there a way to adapt the reasoning process towards the usage of existential quantification?

In Section 3.2 we introduced DatalogMTL$^{\exists}$, an extension of DatalogMTL, with its natural and uniform semantics. We obtained for the studied fragments under OWA only for weakly acyclic programs a decidability result, which is 2-EXPSPACE-complete. We provided for this fragment an algorithm by introducing Skolem terms which map existential quantified variables to the same constant for all time instants.

The next research question addressed the missing *evaluation* scenarios for reasoning in knowledge graphs, combining temporal reasoning with the use of aggregation, existential quantification, and full recursion over cyclic graph structures.

Research Question 3. What queries should a benchmark for DatalogMTL include? What datasets should be used for the benchmark?

In Chapter 4 we observed that state-of-the-art reasoners are not capable of providing a rich feature set to test a combination of required features in knowledge graph systems. Thus, we proposed a benchmark generator which targets exactly those scenarios where a benchmark on the combination of the multiple features is required. That is, instead of providing datasets (which we among others generated during the creation of the generator), we give the community the possibility to generate targeted instances to evaluate a broad range of cases as well as edge cases of their implemented system.

Then, our next research question addressed the implementation and evaluation of the *system* itself by making use of the previously established functionality and evaluation toolset.

Research Question 4. How is the performance of DatalogMTL in comparison with different state-of-the-art systems and between different implementation choices?

In Chapter 5 we discussed the implementation of Temporal Vadalog, our novel reasoning system for knowledge graphs supporting the newest results of DatalogMTL. We evaluated the system regarding state-of-the-art reasoners and showed that our system is capable of (a) supporting temporal aggregation in a reasonable time compared to optimized time-series databases and (b) outperforming state-of-the-art reasoners supporting DatalogMTL.

Finally, our last research question addressed as a concrete *application* on top of the established system the creation of smart contracts with DatalogMTL.

Research Question 5. Is it possible to encode a variety of typical smart contract use cases in DatalogMTL? How can we enable DatalogMTL-based smart contracts on major blockchain platforms?

> In Chapter 6 we highlighted the importance of the temporal domain and developed a smart contract language based on DatalogMTL that complies with the principles of blockchain transactions while maintaining the possibility to reason with DatalogMTL. We showed the applicability of this language by providing case studies and encodings of multiple patterns. Furthermore, we introduced a first version of a translation engine that is capable of making this language accessible to Ethereum, one of the currently most widely used blockchain technologies.

7.2 Future Work

This section reviews some of the remaining open points for future research in multiple dimensions.

Theoretical Contributions. While this work mainly focused on enabling reasoning with DatalogMTL in the setting of knowledge graphs, we encountered several points that require further theoretical research. As seen in the application scenario, we used a restricted form of arithmetic in the programs, which has not been studied for DatalogMTL. Similarly, theoretical results are required for enabling stratified negation along the time axis. Yet, the biggest challenge of DatalogMTL is its data complexity. Using just the past diamond operator with an interval of $[1, 1]$ would already allow to generate programs that are PSPACE-hard in data complexity. While there is ongoing research in the direction of finding tractable (i.e., PTIME) fragments, the currently available fragments are quite limited as either a restricted single operator is allowed or some form of temporal-acyclicity is required. Furthermore, as mentioned in Chapter 5 the chase of DatalogMTL is not terminating in all cases, requiring finding periodic patterns. While we achieved some first results in this regard [BNS21b], further insights are required for efficiently detecting such patterns for full DatalogMTL.

System Design. The integration of temporal reasoning does not end with the support of DatalogMTL. On the one hand, the implementation itself can be improved, for example by optimizing the algorithms such as for joins, or by using optimized data structures for the temporal domain. On the other hand, the temporal features can be extended to go beyond DatalogMTL. For example, one can add support for Allen interval relations, allowing to provide constraints on the relationship of data, where one possible approach is to consider the work by Kontchakov et al. [KPP+16] as a starting point. Another line of interesting future work is the extension of the subject of study to spatial-temporal Datalog queries.

Benchmarks. The extension of spatial-temporal Datalog queries also raises an interesting possibility for extending the benchmark generator with even more sophisticated features such as spatial data. In addition, it would be also interesting to integrate iWarded [ABBS22], a generator targeting existential rules in Vadalog, and extend it to the temporal domain for generating advanced temporal existential rules. Moreover, it would be interesting to also provide a data generator that is directly operating on the input nodes, providing a more real-world oriented generation approach to the users.

Applications. Vadalog is known for its efficient reasoning over multiple financial use cases [BFGS19, BBC+20, BBB+22]. While these use cases work on a single snapshot of the data, it is of high interest to extend these applications to continuous time (and not a collection of snapshots) to make full use of the reasoning capabilities provided by DatalogMTL, providing new insights on the performance of real-world scenarios. Also, the application scenario considered in the thesis, namely using DatalogMTL as a smart contract specification language requires some future work. On the one hand, our results of the translation engine are only preliminary and have to be tested against broader scenarios and extended to full DatalogMTL. On the other hand, it is interesting, especially for DeFi applications, to explore and reason on the insights given by the smart contracts in the analysis of transaction data, since just the reasoning over transaction data in a knowledge graph has shown promising results [BGNS21]. Furthermore, it would be fascinating to study the framework around the creation of a traditional contract by strengthening the interdisciplinary collaboration between jurisprudence and computer science.

List of Figures

List of Tables

List of Algorithms

Bibliography

[10121] What is dlt (distributed ledger technology) ? https://101blockchains.com/what-is-dlt/, 2021. [Online; accessed: 2021-12-20].

[ABBS22] Paolo Atzeni, Teodoro Baldazzi, Luigi Bellomarini, and Emanuel Sallinger. iwarded: A versatile generator to benchmark warded datalog+/- reasoning. In *RuleML+RR*, volume 13752 of *Lecture Notes in Computer Science*, pages 113–129. Springer, 2022.

[ABBV18] Antoine Amarilli, Michael Benedikt, Pierre Bourhis, and Michael Vanden Boom. Query answering with transitive and linear-ordered data. *J. Artif. Intell. Res.*, pages 191–264, 2018.

[ABI+20] Paolo Atzeni, Luigi Bellomarini, Michela Iezzi, Emanuel Sallinger, and Adriano Vlad. Augmenting logic-based knowledge graphs: The case of company graphs. In *KR4L@ECAI*, volume 3020 of *CEUR Workshop Proceedings*, pages 22–27. CEUR-WS.org, 2020.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *VLDB*, pages 480–491. Morgan Kaufmann, 2004.

[ACGR12] Mohammed Al-Kateb, Alain Crolotte, Ahmad Ghazal, and Linda Rose. Adding a temporal dimension to the TPC-H benchmark. In *TPCTC*, volume 7755 of *Lecture Notes in Computer Science*, pages 51–59. Springer, 2012.

[ACLS20] Ralph Abboud, İsmail İlkan Ceylan, Thomas Lukasiewicz, and Tommaso Salvatori. Boxe: A box embedding model for knowledge base completion. In *NeurIPS*, 2020.

[AF05] Alessandro Artale and Enrico Franconi. Temporal description logics. In *Handbook of Temporal Reasoning in Artificial Intelligence*, pages 375–388, 2005.

[AGM15] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. Citybench: A configurable benchmark to evaluate RSP engines using smart city datasets. In *International Semantic Web Conference (2)*, volume 9367 of *Lecture Notes in Computer Science*, pages 374–389. Springer, 2015.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AM89] Martín Abadi and Zohar Manna. Temporal logic programming. *J. Symb. Comput.*, 8(3):277–295, 1989.

[Anw18] Hasib Anwar. Smart contracts: The ultimate guide for the beginners. `https://101blockchains.com/smart-contracts/`, 2018. [Online; accessed: 2021-12-20].

[AvM17] Maher Alharby and Aad van Moorsel. Blockchain-based smart contracts: A systematic mapping study. *CoRR*, abs/1710.06372, 2017.

[BAH19] Ivana Balazevic, Carl Allen, and Timothy M. Hospedales. Hypernetwork knowledge graph embeddings. In *ICANN (Workshop)*, volume 11731 of *Lecture Notes in Computer Science*, pages 553–565. Springer, 2019.

[BBB+22] Luigi Bellomarini, Lorenzo Bencivelli, Claudia Biancotti, Livia Blasi, Francesco Paolo Conteduca, Andrea Gentili, Rosario Laurendi, Davide Magnanimi, Michele Savini Zangrandi, Flavia Tonelli, Stefano Ceri, Davide Benedetto, Markus Nissl, and Emanuel Sallinger. Reasoning on company takeovers: From tactic to strategy. *Data Knowl. Eng.*, 141:102073, 2022.

[BBC+20] Luigi Bellomarini, Marco Benedetti, Stefano Ceri, Andrea Gentili, Rosario Laurendi, Davide Magnanimi, Markus Nissl, and Emanuel Sallinger. Reasoning on company takeovers during the COVID-19 crisis with knowledge graphs. In *RuleML+RR, Industrial*, volume 2644 of *CEUR Workshop Proceedings*, pages 145–156. CEUR-WS.org, 2020.

[BBGS20] Luigi Bellomarini, Davide Benedetto, Georg Gottlob, and Emanuel Sallinger. Vadalog: A modern architecture for automated reasoning with large knowledge graphs. *Inf. Syst.*, page 101528, 2020.

[BBNS22] Luigi Bellomarini, Livia Blasi, Markus Nissl, and Emanuel Sallinger. The temporal vadalog system. In *RuleML+RR*, volume 13752 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2022.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[BDEF15] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *AAAI*, pages 1431–1438. AAAI Press, 2015.

[Ber19] Michael K. Bergman. A common sense view of knowledge graphs. `https://www.mkbergman.com/2244/a-common-sense-view-of-knowledge-graphs/`, 2019. [Online; accessed: 2022-10-21].

[BFGS19] Luigi Bellomarini, Daniele Fakhoury, Georg Gottlob, and Emanuel Sallinger. Knowledge graphs and enterprise AI: the promise of an enabling technology. In *ICDE*, pages 26–37. IEEE, 2019.

[BGJ06] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2006.

[BGNS21] Luigi Bellomarini, Giuseppe Galano, Markus Nissl, and Emanuel Sallinger. Rule-based blockchain knowledge graphs: Declarative AI for solving industrial blockchain challenges. In *RuleML+RR, Industrial*, volume 2956 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021.

[BGO+96] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.

[BGPS17] Luigi Bellomarini, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. Swift logic for big data and knowledge graphs. In *IJCAI*, pages 2–10. ijcai.org, 2017.

[BGZ+20] Lawrence Benson, Philipp M. Grulich, Steffen Zeuch, Volker Markl, and Tilmann Rabl. Disco: Efficient distributed window aggregation. In *EDBT*, pages 423–426. OpenProceedings.org, 2020.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-oriented software architecture, 4th Edition*. Wiley, 2007.

[BJW00] Claudio Bettini, Sushil Jajodia, and Xiaoyang Sean Wang. *Time granularities in databases, data mining, and temporal reasoning*. Springer, 2000.

[BKK+17a] Sebastian Brandt, Elem Güzel Kalayci, Roman Kontchakov, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Ontology-based data access with a Horn fragment of metric temporal logic. In *AAAI*, pages 1070–1076. AAAI Press, 2017.

[BKK+17b] Sebastian Brandt, Elem Güzel Kalayci, Roman Kontchakov, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Ontology-based data access with a horn fragment of metric temporal logic. In *AAAI*, pages 1070–1076. AAAI Press, 2017.

[BKM+17] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. Benchmarking the chase. In *Proc. of PODS*, pages 37–52. ACM, 2017.

[BKR+18] Sebastian Brandt, Elem Güzel Kalayci, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Querying log data with metric temporal logic. *J. Artif. Intell. Res.*, 62:829–877, 2018.

[BMNS20] Luigi Bellomarini, Davide Magnanimi, Markus Nissl, and Emanuel Sallinger. Neither in the programs nor in the data: Mining the hidden financial knowledge with knowledge graphs and reasoning. In *MIDAS@PKDD/ECML*, volume 12591 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2020.

[BNS21a] Luigi Bellomarini, Markus Nissl, and Emanuel Sallinger. Monotonic aggregation for temporal datalog. In *RuleML+RR, Rule Challenge*, volume 2956 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021.

[BNS21b] Luigi Bellomarini, Markus Nissl, and Emanuel Sallinger. Query evaluation in datalogmtl - taming infinite query results. *CoRR*, abs/2109.10691, 2021.

[BNS22] Luigi Bellomarini, Markus Nissl, and Emanuel Sallinger. itemporal: An extensible generator of temporal benchmarks. In *ICDE*, pages 2021–2033. IEEE, 2022.

[BSG18] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. The vadalog system: Datalog-based reasoning for knowledge graphs. *Proc. VLDB Endow.*, 11(9):975–987, 2018.

[BUG+13] Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *NIPS*, pages 2787–2795, 2013.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *SIGMOD Workshop, Vol. 1*, pages 249–264. ACM, 1974.

[CDSS22] Pedro Cabalar, Martín Diéguez, Torsten Schaub, and Anna Schuhmann. Metric temporal answer set programming over timed traces. In *LPNMR*, volume 13416 of *Lecture Notes in Computer Science*, pages 117–130. Springer, 2022.

[CGP11] Andrea Calì, Georg Gottlob, and Andreas Pieris. New expressive languages for ontological query answering. In *Proc. of AAAI*, volume 2011, 2011.

[Che76] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[Cho90] Jan Chomicki. Polynomial time query processing in temporal deductive databases. In *PODS*, pages 379–391. ACM Press, 1990.

[CLMS21] Shivani Choudhary, Tarun Luthra, Ashima Mittal, and Rajat Singh. A survey of knowledge graph embedding and their applications. *CoRR*, abs/2107.07842, 2021.

[CMMO19] Giovanni Ciatto, Alfredo Maffi, Stefano Mariani, and Andrea Omicini. Smart contracts are more than objects: Pro-activeness on the blockchain. In *BLOCKCHAIN*, volume 1010 of *Advances in Intelligent Systems and Computing*, pages 45–53, 2019.

[Dee] DeepAI. What is a knowledge graph? `https://deepai.org/machine-learning-glossary-and-terms/knowledge-graph`. [Online; accessed: 2022-10-21].

[DMSR18] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *AAAI*, pages 1811–1818. AAAI Press, 2018.

[DNR08] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In Maurizio Lenzerini and Domenico Lembo, editors, *Proc. of PODS*, pages 149–158. ACM, 2008.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[EK20] Thomas Eiter and Rafael Kiesel. Weighted LARS for quantitative stream reasoning. In *ECAI*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 729–736. IOS Press, 2020.

[EMSW14] Patrick Ernst, Cynthia Meng, Amy Siu, and Gerhard Weikum. Knowlife: A knowledge graph for health and life sciences. In *ICDE*, pages 1254–1257. IEEE Computer Society, 2014.

[Eth18] Ethereum. Erc20 contract. `https://github.com/ethereum/ethereum-org/blob/master/solidity/token-erc20.sol`, 2018. [Online; accessed: 2022-06-05].

[Eth22] Ethereum. Common patterns. https://docs.soliditylang.org/en/v0.8.15/common-patterns.html, 2022. [Online; accessed: 2022-06-28].

[EW16] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. In *SEMANTiCS (Posters, Demos, SuCCESS)*, volume 1695 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[FGG+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD Conference*, pages 1433–1445. ACM, 2018.

[FKMP05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[FN16] Christopher Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *FAS*W@SASO/ICCAC*, pages 210–215, 2016.

[Gar21] James Garson. Modal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2021 edition, 2021.

[GBJ18] Johann Gamper, Michael H. Böhlen, and Christian S. Jensen. Temporal aggregation. In *Encyclopedia of Database Systems (2nd ed.)*. Springer, 2018.

[GFAA03] Irène Guessarian, Eugénie Foustoucos, Theodore Andronikos, and Foto N. Afrati. On temporal logic versus datalog. *Theor. Comput. Sci.*, 303(1):103–133, 2003.

[GGV02] Georg Gottlob, Erich Grädel, and Helmut Veith. Datalog LITE: a deductive query language with linear time model checking. *ACM Trans. Comput. Log.*, 3(1):42–79, 2002.

[GHLZ13] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Found. Trends Databases*, 5(2):105–195, 2013.

[GJO16] Víctor Gutiérrez-Basulto, Jean Christoph Jung, and Ana Ozaki. On metric temporal description logics. In *ECAI*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 837–845. IOS Press, 2016.

[GLP14] Georg Gottlob, Thomas Lukasiewicz, and Andreas Pieris. Datalog+/-: Questions and answers. In *KR*. AAAI Press, 2014.

[GM93] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[GM10] Claudio Gallicchio and Alessio Micheli. Graph echo state networks. In *IJCNN*, pages 1–8. IEEE, 2010.

[GO22] Ricardo Guimarães and Ana Ozaki. Reasoning in knowledge graphs (invited paper). In *AIB*, volume 99 of *OASIcs*, pages 2:1–2:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[GOPS12] Georg Gottlob, Giorgio Orsi, Andreas Pieris, and Mantas Simkus. Datalog and its extensions for semantic web databases. In *Reasoning Web*, volume 7487 of *Lecture Notes in Computer Science*, pages 54–77. Springer, 2012.

[GP15] Georg Gottlob and Andreas Pieris. Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue. In *IJCAI*, pages 2999–3007. AAAI Press, 2015.

[GR22] Valentin Goranko and Antje Rumberg. Temporal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2022 edition, 2022.

[GTHS15] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Fast rule mining in ontological knowledge bases with AMIE+. *VLDB J.*, 24(6):707–730, 2015.

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[HAF14] Bardh Hoxha, Houssam Abbas, and Georgios Fainekos. Benchmarks for temporal logic requirements for automotive systems. In *ARCH@CPSWeek*, volume 34 of *EPiC Series in Computing*, pages 25–30. EasyChair, 2014.

[HB16] Mike Hearn and Richard Gendal Brown. Corda: A distributed ledger. *Corda Technical White Paper*, 2016.

[HQC+21] Yuanzhe Hao, Xiongpai Qin, Yueguo Chen, Yaru Li, Xiaoguang Sun, Yu Tao, Xiao Zhang, and Xiaoyong Du. Ts-benchmark: A benchmark for time series databases. In *ICDE*, pages 588–599. IEEE, 2021.

[HYL17] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[HZ18] Jingwen Hu and Yong Zhong. A method of logic-based smart contracts for blockchain system. In *ICDPA*, pages 58–61, 2018.

[IGRS16] Florian Idelberger, Guido Governatori, Régis Riveret, and Giovanni Sartor. Evaluation of logic-based smart contracts for blockchain systems. In *RuleML*, volume 9718 of *Lecture Notes in Computer Science*, pages 167–183, 2016.

[JPC+22] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S. Yu. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Trans. Neural Networks Learn. Syst.*, 33(2):494–514, 2022.

[Kau22] Henry A. Kautz. The third AI summer: AAAI robert s. engelmore memorial lecture. *AI Mag.*, 43(1):93–104, 2022.

[KBC+19] Elem Güzel Kalayci, Sebastian Brandt, Diego Calvanese, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Ontology-based access to temporal data with ontop: A framework proposal. *Int. J. Appl. Math. Comput. Sci.*, 29(1):17–30, 2019.

[KEKE19] Elmar Kiesling, Andreas Ekelhart, Kabul Kurniawan, and Fajar J. Ekaputra. The SEPSES knowledge graph: An integrated resource for cybersecurity. In *ISWC (2)*, volume 11779 of *Lecture Notes in Computer Science*, pages 198–214. Springer, 2019.

[KFM+13] Martin Kaufmann, Peter M. Fischer, Norman May, Andreas Tonder, and Donald Kossmann. Tpc-bih: A benchmark for bitemporal databases. In *TPCTC*, volume 8391 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2013.

[KFM+15] Martin Kaufmann, Peter M. Fischer, Norman May, Chang Ge, Anil K. Goel, and Donald Kossmann. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *ICDE*, pages 471–482. IEEE Computer Society, 2015.

[KGKH20] Mark Kaminski, Bernardo Cuenca Grau, Egor V. Kostylev, and Ian Horrocks. Complexity and expressive power of disjunction and negation in limit datalog. In *AAAI*, pages 2862–2869, 2020.

[KMV+13] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD Conference*, pages 1173–1184. ACM, 2013.

[KNP+22] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. Convergence of datalog over (pre-) semirings. In *PODS*, pages 105–117. ACM, 2022.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, 2(4):255–299, 1990.

[KP18] Seyed Mehran Kazemi and David Poole. Simple embedding for link prediction in knowledge graphs. In *NeurIPS*, pages 4289–4300, 2018.

[KPP+16] Roman Kontchakov, Laura Pandolfo, Luca Pulina, Vladislav Ryzhikov, and Michael Zakharyaschev. Temporal and spatial OBDA with many-dimensional halpern-shoham logic. In *IJCAI*, pages 1160–1166. IJCAI/AAAI Press, 2016.

[KRWZ18] Stanislav Kikot, Vladislav Ryzhikov, Przemyslaw Andrzej Walega, and Michael Zakharyaschev. On the data complexity of ontology-mediated queries with MTL operators over timed words. In *Description Logics*, volume 2211 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[KS95] Nick Kline and Richard T. Snodgrass. Computing temporal aggregates. In *ICDE*, pages 222–231. IEEE Computer Society, 1995.

[KW17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR (Poster)*. OpenReview.net, 2017.

[LNSW23] Matthias Lanzinger, Markus Nissl, Emanuel Sallinger, and Przemyslaw Andrzej Walega. Temporal datalog with existential quantification. In *IJCAI*, 2023. to appear.

[LW22] Matthias Lanzinger and Przemyslaw Andrzej Walega. Datalog with existential quantifiers and temporal operators (extended abstract). In *Datalog*, volume 3203 of *CEUR Workshop Proceedings*, pages 139–144. CEUR-WS.org, 2022.

[Mar09] Bruno Marnette. Generalized schema-mappings: from termination to tractability. In *Proc. of PODS*, pages 13–22. ACM, 2009.

[MCRS19] Christian Meilicke, Melisachew Wudage Chekol, Daniel Ruffinelli, and Heiner Stuckenschmidt. Anytime bottom-up rule learning for knowledge graph completion. In *IJCAI*, pages 3137–3143. ijcai.org, 2019.

[MMS79] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, 1979.

[MSZ13] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. Extending the power of datalog recursion. *VLDB J.*, 22(4):471–493, 2013.

[MTKW18] David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. Datalog: concepts, history, and outlook. In *Declarative Logic Programming*, pages 3–100. ACM / Morgan & Claypool, 2018.

[Nil86] Nils J. Nilsson. Probabilistic logic. *Artif. Intell.*, 28(1):71–87, 1986.

[NS13] Tu Ngoc Nguyen and Wolf Siberski. SLUBM: an extended LUBM benchmark for stream reasoning. In *OrdRing@ISWC*, volume 1059 of *CEUR Workshop Proceedings*, pages 43–54. CEUR-WS.org, 2013.

[NS22a] Markus Nissl and Emanuel Sallinger. Modelling smart contracts with datalogmtl. In *EDBT/ICDT Workshops*, volume 3135 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[NS22b] Markus Nissl and Emanuel Sallinger. Towards bridging traditional and smart contracts with datalog-based languages. In *Datalog*, volume 3203 of *CEUR Workshop Proceedings*, pages 68–82. CEUR-WS.org, 2022.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[PH17] Danila Piatov and Sven Helmer. Sweeping-based temporal aggregation. In *SSTD*, volume 10411 of *Lecture Notes in Computer Science*, pages 125–144. Springer, 2017.

[Pop18] Serguei Popov. The tangle. *White paper*, 1(3), 2018.

[Que21] QuestDB. Benchmarking database performance with time series workloads. `https://questdb.io/time-series-benchmark-suite/`, 2021. [Online; accessed: 2021-11-04].

[Rey16] Mark Reynolds. Metric temporal logic revisited. *Acta Informatica*, 53(3):301–324, 2016.

[RHT+17] Maya Rotmensch, Yoni Halpern, Abdulhakim Tlimat, Steven Horng, and David Sontag. Learning a health knowledge graph from electronic medical records. *Scientific reports*, 7(1):1–11, 2017.

[RKG+18] Alessandro Ronca, Mark Kaminski, Bernardo Cuenca Grau, Boris Motik, and Ian Horrocks. Stream reasoning in temporal datalog. In *AAAI*, pages 1941–1948. AAAI Press, 2018.

[RS92] Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *PODS*, pages 114–126. ACM Press, 1992.

[Sci20] Edward Sciore. *Database Design and Implementation - Second Edition*. Springer, 2020.

[SD20] Adriana Stancu and Mihaita Dragan. Logic-based smart contracts. In *WorldCIST (1)*, volume 1159 of *Advances in Intelligent Systems and Computing*, pages 387–394, 2020.

[SDNT19] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. Rotate: Knowledge graph embedding by relational rotation in complex space. In *ICLR (Poster)*. OpenReview.net, 2019.

[Sin12] Amit Singhal. Introducing the knowledge graph: things, not strings. `https://blog.google/products/search/introducing-knowledge-graph-things-not/`, 2012. [Online; accessed: 2022-10-21].

[Sin21] Amarpreet Singh. What is decentralized finance or defi explained. `https://medium.com/brandlitic/dc0ce376f7b2`, 2021. [Online; accessed: 2021-12-20].

[Smu95] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[SU19] Dmitrii Suvorov and Vladimir Ulyantsev. Smart contract design meets state machine synthesis: Case studies. *CoRR*, abs/1906.02906, 2019.

[SYZ15] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. In *ICDE*, pages 867–878. IEEE Computer Society, 2015.

[TCWCGK21] David J. Tena Cucala, Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, and Egor V. Kostylev. Stratified negation in datalog with metric temporal operators. In *AAAI*, pages 6488–6495. AAAI Press, 2021.

[Tim] Timescale. Time series benchmark suite (tsbs). `https://github.com/timescale/tsbs`. [Online; accessed: 2021-08-18].

[TWR+16] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2071–2080. JMLR.org, 2016.

[UKE22] Jacopo Urbani, Markus Krötzsch, and Thomas Eiter. Chasing streams with existential rules. *arXiv preprint arXiv:2205.02220*, 2022.

[Ulu19] Dogan Ulus. Timescales: A benchmark generator for MTL monitoring tools. In *RV*, volume 11757 of *Lecture Notes in Computer Science*, pages 402–412. Springer, 2019.

[WCGKK19] Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, Mark Kaminski, and Egor V. Kostylev. Datalogmtl: Computational complexity and expressive power. In *IJCAI*, pages 1886–1892. ijcai.org, 2019.

[WCGKK20a] Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, Mark Kaminski, and Egor V. Kostylev. Datalogmtl over the integer timeline. In *KR*, pages 768–777, 2020.

[WCGKK20b] Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, Mark Kaminski, and Egor V. Kostylev. Tractable fragments of datalog with metric temporal operators. In *IJCAI*, pages 1919–1925. ijcai.org, 2020.

[WHC+19] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. KGAT: knowledge graph attention network for recommendation. In *KDD*, pages 950–958. ACM, 2019.

[WHWCG22] Dingmin Wang, Pan Hu, Przemyslaw Andrzej Walega, and Bernardo Cuenca Grau. Meteor: Practical reasoning in datalog with metric temporal operators. In *AAAI*, pages 5906–5913. AAAI Press, 2022.

[WKCG19] Przemyslaw Andrzej Walega, Mark Kaminski, and Bernardo Cuenca Grau. Reasoning over streaming data in metric temporal datalog. In *AAAI*, pages 3092–3099. AAAI Press, 2019.

[WKN+22] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with progam synthesis. In *SIGMOD Conference*, pages 79–93. ACM, 2022.

[WPC+21] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Trans. Neural Networks Learn. Syst.*, 32(1):4–24, 2021.

[WQW21] Meihong Wang, Linling Qiu, and Xiaoli Wang. A survey on knowledge graph embeddings for link prediction. *Symmetry*, 13(3):485, 2021.

[WTCKCG21] Przemyslaw Andrzej Walega, David J. Tena Cucala, Egor V. Kostylev, and Bernardo Cuenca Grau. Datalogmtl with negation under stable models semantics. In *KR*, pages 609–618, 2021.

[WWCG22] Dingmin Wang, Przemyslaw Andrzej Walega, and Bernardo Cuenca Grau. Seminaive materialisation in datalogmtl. *CoRR*, abs/2208.07100, 2022.

[WZCG21] Przemyslaw Andrzej Walega, Michal Zawidzki, and Bernardo Cuenca Grau. Finitely materialisable datalog programs with metric temporal operators. In *KR*, pages 619–628, 2021.

[WZW+20] Qiange Wang, Yanfeng Zhang, Hao Wang, Liang Geng, Rubao Lee, Xiaodong Zhang, and Ge Yu. Automating incremental and asynchronous evaluation for recursive aggregate data processing. In *SIGMOD Conference*, pages 2439–2454. ACM, 2020.

[YKS+22] Zi Ye, Yogan Jaya Kumar, Goh Ong Sing, Fengyan Song, and Junsong Wang. A comprehensive survey of graph neural networks for knowledge graphs. *IEEE Access*, 10:75729–75741, 2022.

[YOO+13] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William K. Robertson, Ari Juels, and Engin Kirda. Beehive: large-scale log analysis for detecting suspicious activity in enterprise networks. In *ACSAC*, pages 199–208. ACM, 2013.

[YW03] Jun Yang and Jennifer Widom. Incremental computation and maintenance of temporal aggregates. *VLDB J.*, 12(3):262–283, 2003.

[YYH+15] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. In *ICLR (Poster)*, 2015.

[YYR+18] Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 5694–5703. PMLR, 2018.

[Zan12] Carlo Zaniolo. Logical foundations of continuous query languages for data streams. In *Datalog*, volume 7494 of *Lecture Notes in Computer Science*. Springer, 2012.

[ZAO93] Carlo Zaniolo, Natraj Arni, and KayLiang Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *DOOD*, volume 760 of *LNCS*, pages 204–221. Springer, 1993.

[ZMT+01] Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. Efficient computation of temporal aggregates with range predicates. In *PODS*. ACM, 2001.

[ZPCC12] Ying Zhang, Minh-Duc Pham, Óscar Corcho, and Jean-Paul Calbimonte. Srbench: A streaming RDF/SPARQL benchmark. In *International Semantic Web Conference (1)*, volume 7649 of *Lecture Notes in Computer Science*, pages 641–657. Springer, 2012.