The Temporal Vadalog System: Temporal Datalog-Based Reasoning

LUIGI BELLOMARINI

Bank of Italy, Italy
(*e-mail:* luigi.bellomarini@bancaditalia.it)

LIVIA BLASI

Bank of Italy, Italy
TU Wien, Austria
(*e-mail:* livia.blasi@bancaditalia.it)

MARKUS NISSL

TU Wien, Austria
(*e-mail:* nissl@dbai.tuwien.ac.at)

EMANUEL SALLINGER

TU Wien, Austria
University of Oxford, UK
(*e-mail:* sallinger@dbai.tuwien.ac.at)

Abstract

In the wake of the recent resurgence of the Datalog language of databases, together with its extensions for ontological reasoning settings, this work aims to bridge the gap between the theoretical studies of DatalogMTL (Datalog extended with metric temporal logic) and the development of production-ready reasoning systems. In particular, we lay out the functional and architectural desiderata of a modern reasoner and propose our system, Temporal Vadalog. Leveraging the vast amount of experience from the database community, we go beyond the typical chase-based implementations of reasoners, and propose a set of novel techniques and a system that adopts a modern data pipeline architecture. We discuss crucial architectural choices, such as how to guarantee termination when infinitely many time intervals are possibly generated, how to merge intervals, and how to sustain a limited memory footprint. We discuss advanced features of the system, such as the support for time series, and present an extensive experimental evaluation. This paper is a substantially extended version of "The Temporal Vadalog System" as presented at RuleML+RR '22.

KEYWORDS: temporal reasoning, DatalogMTL, Datalog, Vadalog

1 Introduction

In recent years, the Datalog language (Ceri et al. 1989) has been experiencing renewed interest, thanks to the growing adoption for *Knowledge Representation and Reasoning* applications (Gottlob 2022). The additional requirements introduced by reasoning spawned prolific research towards new extensions of the language to support advanced features such as existential quantification and aggregation. These extensions are frequently shared under the common name of Datalog$^\pm$ (Calì et al. 2012) and include languages, technically, "fragments," that exhibit a very good tradeoff between computational complexity and expressive power, such as Warded and Shy Datalog$^\pm$ (Baldazzi et al. 2022).

As the new logical languages flourish, the expectations for declarative, explainable expressivity of complex domains, jointly with feasible computation, are rekindling the idea of using deductive languages in practical applications, with modern systems (Leone et al. 2019) based on Datalog in production settings, often in combination with knowledge-based AI models, such as *knowledge graphs* (Bellomarini et al. 2019).

With so many real-world adoptions, the temporal perspective is increasingly becoming a first-class requirement of Datalog. In fact, with the pervasive constant and voluminous streams of data, we are witnessing that temporal reasoning – the ability to reason about events and intervals in time – is becoming of the essence in many fields, from healthcare to finance, robotics to transportation. We can look at a plethora of use cases: the longitudinal studies of anticancer treatments to explain and prevent adverse events (Ma et al. 2021), the analysis and prediction of stock market time series (Walega et al. 2019), the tracking of objects for unmanned vehicles (Lin et al. 2022), the processing of IoT devices interactions (Walega et al. 2019), and the exploitation of log data (Brandt et al. 2018).

The proposal for an extension of Datalog with *Metric Temporal Logic* operators, or DatalogMTL (Brandt et al. 2018), shows an extremely promising impulse towards a declarative language with sufficient expressive power to handle reasoning settings, such as recursion, while offering time awareness within a computationally effective framework.

Positioning of the paper

This paper studies the theoretical and practical underpinnings at the foundation of modern implementations of DatalogMTL. We illustrate the functional and architectural desiderata of such systems and present the *Temporal Vadalog* system, a fully-engineered reasoning DatalogMTL tool. This work is a substantially extended version of a RuleML+RR '22 paper (Bellomarini et al. 2022) introducing the system. Besides proposing a broader and completely reformulated presentation of the content, the contribution is here enriched with (i) an in-depth discussion about the logical optimization of the temporal operators; (ii) the implementation of new temporal operators (namely, *since* and *until*); (iii) an entirely new part dedicated to the implementation of time series in Temporal Vadalog; (iv) a corpus of new experiments.

To present the ideas at the foundation of Temporal Vadalog, let us start introducing DatalogMTL with a financial use case example from our work with the Bank of Italy.

Example 1.1.

Changes in strategically important companies may face government scrutiny to uphold regulatory standards, focusing on both the companies and their new shareholders. We describe the scenario by a database $\mathcal{D}$ of facts about company shares and the following set Π of DatalogMTL rules.

$$\boxplus_{[0,1]} significantShare(X, Y),$$
$$\neg \diamondsuit_{[0,1]} significantShare(X, Y) \to significantOwner(X, Y) \tag{1}$$

$$watchCompany(Y), significantOwner(X, Y),$$
$$connected(X, Z) \to watchCompany(Z) \tag{2}$$

We assume the reader is familiar with Datalog rule syntax: the left-hand side of the rule (the body) is the implication premise and is composed of a logical conjunction of predicates over terms, that is constants or variables; the right-hand side (the head) is the implication conclusion. In the rules, we find predicates from the relational schema of $\mathcal{D}$. In particular, significantShare holds whenever X retains a high number of shares of Y; significantOwner is then a derived, namely, "intensional" predicate, obtained by Rule 1. Then, by Rule 2, whenever Y is interested in a watchCompany fact to indicate that the authority deems that company of some strategic relevance, given that X is a significant owner of Y, then all companies Z connected to X will be watched over as well. Neglecting for a moment the temporal angle of Π, its semantics is straightforward: whenever the premise of one rule holds with respect to $\mathcal{D}$, then $\mathcal{D}$ is augmented with new facts to satisfy the implication if it does not already hold. For instance, if $\mathcal{D}$ contains the facts watchCompany(ACME), significantOwner(A, ACME), connected(A, EMCA), then the new fact watchCompany(EMCA) will be added to $\mathcal{D}$. Here, the temporal perspective is introduced by the $\boxplus$ and $\diamondsuit$ operators. When prefixed with $\boxplus$, a predicate holds, only if the predicate itself continuously holds in the interval ($[t, t+1]$, if Rule 1 is evaluated at t) in the future, as indicated by the operator internal in the index. The $\diamondsuit$ operator modifies a predicate's semantics so that it holds if the predicate itself holds at least once in the past interval ($[t-1, t]$, if Rule 1 is evaluated at t). Clearly, in a temporal interpretation of Datalog, all facts, including the "extensional" ones of $\mathcal{D}$, are time-annotated. In the example, Rule 1 fires only if X, who has never been a significant shareholder for Y in the past, becomes such in the future.

More formally speaking, the semantics of a set of Datalog rules Π is defined via the chase procedure (Maier et al. 1979), which modifies $\mathcal{D}$ as long as all the rules of Π are satisfied. Multiple variants of the chase arose, including many supporting DatalogMTL. At the same time, several DatalogMTL fragments were also introduced, to carefully balance expressive power and computational complexity: DatalogMTL$^{\text{FP}}$ (Walega et al. 2019) and its core and linear eponymous DatalogMTL$^{\diamondsuit}_{core}$ and DatalogMTL$^{\diamondsuit}_{lin}$ (Walega et al. 2020b), or Integer DatalogMTL (Walega et al. 2020a) offer a good prospect for reasoning over time, with some of them being tractable, that is *query answering* can be evaluated in polynomial time with respect to the database size. Moreover, the system should support operations over numeric values and aggregations, as well as negation.

To get to the core of our discussion, let us lay out the characteristics that a reasoning system should support to deal with time as a first-class system, via DatalogMTL.

Functional desiderata

A temporal reasoning system is a "reasoner" in the first place. A good yardstick for the features it should incorporate then comes from the experience with *knowledge graph management systems*: such tools should adopt a simple, modular, low-complexity, and highly expressive language. DatalogMTL fragments like the ones mentioned above should then be supported. To cope with real-world applications, monotonic aggregations should be supported as well (Bellomarini et al. 2021a).

Architectural desiderata

The existing systems implementing DatalogMTL adopt time-aware extensions of the chase. Yet, the vast experience from the database community suggests that a direct implementation of the chase can be a suboptimal choice, affected by multiple limitations. For efficiency and memory footprint, the extensional database and all intermediate facts must be in memory. We wish for reasoning systems to be memory-bound, using a modern data processing pipeline architecture. As for desiderata specific to temporal reasoners, temporal programs can produce facts that periodically hold over time, with repeating patterns potentially generating infinite facts, which practically can be represented compactly depending on the DatalogMTL fragment. A temporal reasoner should recognize these fragments, identify the patterns at runtime, and guarantee termination. Architecturally, it should have a configurable toolbox of strategies for different fragments. Time intervals should be carefully handled, with a selection of algorithms to merge and compress them for a space-efficient representation.

The development of temporal reasoners is in its infancy, with the only proposals for practical implementation of DatalogMTL being either not engineered for production use or not supporting essential features such as recursion, like in the system by Brandt et al. (2017). Conversely, MeTeoR (Wang et al. 2022) supports recursion; still, it does not offer operations over numeric values or aggregations.

Contribution

In this paper, we present the Temporal Vadalog system, a fully-engineered reasoning system that addresses the above desiderata. Functionally, it offers:

- support for **multiple DatalogMTL fragments**, including DatalogMTL^{FP} (i.e. *forward-propagating*; resp. BP, i.e. *backward-propagating*) over the real timeline, and DatalogMTL^{FP} (resp. BP) over the integer timeline
- native support of the **DatalogMTL operators** (box, diamond, since and until)
- operations over **numeric** values
- **recursion**, **aggregations**, and **stratified negation**
- **time series** and time series operators.

Its architecture is based on a modern data pipeline style, where programs are evaluated in a *pull-based* and *query-driven* manner, sustaining limited memory footprint and high scalability for medium- to high-size settings. More in particular:

- **rewriting-based optimization**
- fragment-aware **termination control strategies**
- seamless integration of **temporal and non-temporal reasoning**.

Overview

Section 2 presents DatalogMTL and an overview of time series operators. Section 3 covers the main contribution of the paper, detailing the Temporal Vadalog pipeline and its components; Section 4 shows how the system can be used to reason over time series. Section 5 presents the empirical evaluation of the system with a discussion of the experiments. In Section 6 we discuss the related work; finally, we conclude the paper in Section 7. Additional material can be found in the supplementary material.

2 Preliminaries

2.1 DatalogMTL

DatalogMTL is Datalog extended with operators from the metric temporal logic. This section is a summary of DatalogMTL with stratified negation under continuous semantics.

DatalogMTL is defined over the rational timeline, that is an ordered set of rational numbers $\mathbb{Q}$. An interval $\varrho = \langle \varrho^-, \varrho^+ \rangle$ is a non-empty subset of $\mathbb{Q}$ such that for each $t \in \mathbb{Q}$ where $\varrho^- < t < \varrho^+$, $t \in \varrho$, and the endpoints $\varrho^-, \varrho^+ \in \mathbb{Q} \cup \{-\infty, \infty\}$. The brackets denote whether the interval is closed ("[]"), half-open ("[)","(]") or open ("()"), whereas angle brackets ("$\langle\rangle$") are used when unspecified. An interval is *punctual* if it is of the form $[t,t]$, *positive* if $\varrho^- \geq 0$, and *bounded* if $\varrho^-, \varrho^+ \in \mathbb{Q}$.

DatalogMTL extends the syntax of Datalog with negation with temporal operators (Tena Cucala et al. 2021). For the following definitions, we consider a function-free first-order signature. An *atom* is of the form $P(\boldsymbol{\tau})$, where P is a n-ary predicate and $\boldsymbol{\tau}$ is a n-ary tuple of terms, where a term is either a constant or a variable. An atom is *ground* if it contains no variables. A *fact* is an expression $P(\boldsymbol{\tau})@\varrho$, where ϱ is an interval and $P(\boldsymbol{\tau})$ a ground atom and a *database* is a set of facts. A *literal* is an expression given by the following grammar, where ϱ is a positive interval: $A ::= \top \mid \bot \mid P(\boldsymbol{\tau}) \mid \boxminus_\varrho A \mid \boxplus_\varrho A \mid \diamondsuit\!\!\!\!-_\varrho A \mid \diamondsuit\!\!\!\!+_\varrho A \mid A \, \mathcal{S}_\varrho \, A \mid A \, \mathcal{U}_\varrho \, A$. A *rule* is an expression given by the following grammar, where $i, j \geq 0$, each A_k $(k \geq 0)$ is a literal and B is an atom: $A_1 \wedge \cdots \wedge A_i \wedge \mathrm{not}\, A_{i+1} \wedge \cdots \wedge \mathrm{not}\, A_{i+j} \rightarrow B$. The conjunction of literals A_k is the rule body, where $A_1 \wedge \cdots \wedge A_i$ denote positive literals and $A_{i+1} \wedge \cdots \wedge A_{i+j}$ denote negated (i.e. prefixed with *not*) literals. The atom B is the rule head. A rule is *safe* if each variable occurs in at least one positive body literal, *positive* if it has no negated body literals (i.e. $j = 0$), and *ground* if it contains no variables. A program Π is a set of safe rules and is *stratifiable* if there exists a stratification of a program Π. A stratification of Π is defined as a function σ that maps each predicate P in Π to a positive integer (*stratum*) s.t. for each rule, where P^h denotes a predicate of the head, and P^+ (resp. P^-) a positive (negative) body predicate, $\sigma(P^h) \geq \sigma(P^+)$ and $\sigma(P^h) > \sigma(P^-)$.

The semantics of DatalogMTL is given by an interpretation $\mathfrak{M}$ that specifies for each time point $t \in \mathbb{Q}$ and each ground atom $P(\boldsymbol{\tau})$, whether $P(\boldsymbol{\tau})$ is satisfied at t, in which case we write $\mathfrak{M}, t \models P(\boldsymbol{\tau})$. This satisfiability notion extends to ground literals as follows:

$$\mathfrak{M}, t \models \top \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for each } t$$
$$\mathfrak{M}, t \models \bot \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for no } t$$
$$\mathfrak{M}, t \models \boxminus_\varrho A \qquad\qquad\qquad \text{iff } \mathfrak{M}, s \models A \text{ for all } s \text{ with } t - s \in \varrho$$
$$\mathfrak{M}, t \models \boxplus_\varrho A \qquad\qquad\qquad \text{iff } \mathfrak{M}, s \models A \text{ for all } s \text{ with } s - t \in \varrho$$
$$\mathfrak{M}, t \models A \, \mathcal{S}_\varrho \, A' \quad \text{iff } \mathfrak{M}, s \models A' \text{ for some } s \text{ with } t - s \in \varrho \wedge \mathfrak{M}, r \models A \text{ for all } r \in (s, t)$$
$$\mathfrak{M}, t \models A \, \mathcal{U}_\varrho \, A' \quad \text{iff } \mathfrak{M}, s \models A' \text{ for some } s \text{ with } s - t \in \varrho \wedge \mathfrak{M}, r \models A \text{ for all } r \in (t, s)$$
$$\mathfrak{M}, t \models \diamondsuit_\varrho A \qquad\qquad\qquad \text{iff } \mathfrak{M}, s \models A \text{ for some } s \text{ with } t - s \in \varrho$$
$$\mathfrak{M}, t \models \diamondplus_\varrho A \qquad\qquad\qquad \text{iff } \mathfrak{M}, s \models A \text{ for some } s \text{ with } s - t \in \varrho$$

An interpretation $\mathfrak{M}$ satisfies not A ($\mathfrak{M}, t \models \text{not } A$), if $\mathfrak{M}, t \not\models A$, a fact $P(\boldsymbol{\tau})@\varrho$, if $\mathfrak{M}, t \models P(\boldsymbol{\tau})$ for all $t \in \varrho$, and a set of facts $\mathcal{D}$ if it is a model of each fact in $\mathcal{D}$. Furthermore, $\mathfrak{M}$ satisfies a ground rule r if $\mathfrak{M}, t \models A_k$ for $0 \leq k \leq i$ and $\mathfrak{M}, t \models \text{not } A_k$ for $i + 1 \leq k \leq i + j$ for every t; for every t, if the literals in the body are satisfied, so is the head $\mathfrak{M}, t \models B$; $\mathfrak{M}$ satisfies a rule when it satisfies every possible grounding of the rule. Moreover, $\mathfrak{M}$ is a *model* of a program if it satisfies every rule in the program and the program has a stratification, i.e. it is *stratifiable*. Given a stratifiable program Π and a set of facts $\mathcal{D}$, we call $\mathfrak{C}_{\Pi,\mathcal{D}}$ the *canonical model* of Π and $\mathcal{D}$ (Brandt et al. 2018), and define it as the minimum model of Π and $\mathcal{D}$, i.e. $\mathfrak{C}_{\Pi,\mathcal{D}}$ is the minimum model for all the facts of $\mathcal{D}$ and the rules of Π. In this context, "minimum" means that the set of positive literals in $\mathfrak{M}$ is minimized or, equivalently, that the positive literals of this model are contained in every other model. Since Π is stratifiable, this minimum model exists and is unique (Gelder et al. 1991). According to Tena Cucala's notation (Tena Cucala et al. 2021), we say that a stratifiable program Π and a set of facts $\mathcal{D}$ entail a fact $P(\boldsymbol{\tau})@\varrho$, written as $(\Pi, \mathcal{D}) \models P(\boldsymbol{\tau})@\varrho$, if $\mathfrak{C}_{\Pi,\mathcal{D}} \models P(\boldsymbol{\tau})@\varrho$. In the remainder of the paper, we will assume the stratification of programs (or set of rules) as implicit.

In this context, the *query answering* or *reasoning* task is defined as follows: given the pair $Q = (\Pi, Ans)$, where Π is a set of rules, Ans is an n-ary predicate, and the query Q is evaluated over $\mathcal{D}$, then $Q(\mathcal{D})$ is defined as $Q(\mathcal{D}) = \{(\bar{t}, \varrho) \in dom(\mathcal{D})^n \times time(\mathcal{D}) \mid (\Pi, \mathcal{D}) \models Ans(\bar{t})@\varrho\}$, where $\bar{t}$ is a tuple of terms, the domain of $\mathcal{D}$, denoted $dom(\mathcal{D})$, is the set of all constants that appear in the facts of $\mathcal{D}$, and the set of all the time intervals in $\mathcal{D}$ is denoted as $time(\mathcal{D})$. As we shall see in practical cases, the Ans predicate of Π will be sometimes called "query predicate" and provided to the reasoning system with specific conventions, which we omit for space reasons, but will render in textual explanations.

Fragments of DatalogMTL

We define DatalogMTL$^{\text{FP}}$ (Walega et al. 2019) as the DatalogMTL language restricted to only make use of *forward-propagating* operators, that is the operators that only examine the past, as in $\diamondsuit$, $\boxminus$ and $\mathcal{S}$. DatalogMTL$^{\text{BP}}$ is symmetrical to DatalogMTL$^{\text{FP}}$ using backward-propagating operators, for example $\diamondplus$, $\boxplus$ and $\mathcal{U}$. A *linear* language has at most one intensional atom in the body of the rules, while in a *core* language, in addition, rules without $\bot$ in the head contain only one body atom. DatalogMTL$^{\diamondsuit}_{core}$, DatalogMTL$^{\diamondsuit}_{lin}$

(Walega et al. 2020b) are thus defined as the *core* and *linear* fragments using only the ⇔ operator. Finally, Integer DatalogMTL (Walega et al. 2020a) is DatalogMTL operating exclusively on an integer (discrete) timeline.

2.2 Time series

Time series data is a sequence of data points, collected over time, that measure the performance of a particular variable. For example, we may have time series for stock prices or inflation expectations in finance and economics, humidity readings in domotics, and all the relevant vital measurements for healthcare.

Time series are typically made of sequences of evenly-spaced punctual temporal intervals, in which case they are considered *regular*; otherwise, they are called *irregular* time series. The distance between intervals defines the frequency (or time resolution).

To understand how to work with time series, we need a clear overview of the operators that can be applied and how they manipulate and transform data to detect and measure patterns and trends. Let us go through an introduction to the common time series operators, and we will later explore their implementation in Temporal Vadalog.

Shifting

The shifting operator moves the time index of the data, allowing to compare data points from different periods (e.g., the same month in different years).

Rolling

The rolling operator calculates a moving time window (rolling window), typically to apply statistical metrics to smooth out short-term data fluctuations.

Resampling

The resampling operator transforms the time series by changing its frequency, by lowering it (*downsampling)* or by increasing it (*upsampling*).

Moving averages

Moving averages use rolling windows to smooth data and filter noise and fluctuations. Types depend on the different weight over data points, and include: a) *Simple Moving Average*, where all points have equal weight, and the *Centered Moving Average* variant; b) *Exponential Moving Average*, where recent points have more weight.

Stock and flow

Measurements in time series can be stocks, quantities existing at a specific time, or flows, variables measured over an interval of time, comparable to the rate or speed, and can be derived from stocks (and vice versa). For instance, in real estate, the stock could refer to the available housing units, while the flow represents the rate of new units entering it.

For the conversion between measurement units, we use the following operators: a) *Stock to flow*, the difference between stocks over consecutive time points; b) *Flow to stock*, the cumulative sum of a series of flows over time.

Seasonal decomposition

The seasonal decomposition operator splits a time series into its trend, seasonal, and residual components, which helps identify patterns and trends.

3 The Temporal Vadalog Architecture

The architecture aims to support query answering with DatalogMTL: given a database of time-annotated facts and a set of rules with one or more query predicates, we produce an execution plan for that query. Our approach is inspired by our extensive experience in constructing database and knowledge graph management systems. Our architecture builds on that of the Vadalog system (Bellomarini et al. 2018), which in its current version does not support temporal reasoning. Vadalog adopts the *volcano iterator model* (Graefe and McKenna 1993), thus employing consolidated query evaluation techniques from a reasoning perspective. In this work, we go beyond and extend the approach in such a way that the query plans include temporal operators (i.e. they are *time-aware*). Rather than providing a comprehensive taxonomy of all the components in the architecture, we will take a thematic walk-through, with a focus on addressing the challenges of temporal reasoning. In the next section, we will introduce our time-aware execution pipeline.

3.1 A time-aware execution pipeline

Similar to the *pipe and filters* architecture (Buschmann et al. 2007), a DatalogMTL program Π is transformed into an execution pipeline that takes data from the input sources, applies necessary modifications, including relational algebra operations (e.g., projection, selection) or time-based ones, and generates the intended output as a result.

Building the pipeline

Building the pipeline involves four main steps: (i) A *logic optimizer* rewriting tasks to simplify programs and ensure that only combined individual operators are allowed in the canonical form. (ii) A *logic compiler* transforming the rules into in-memory placeholder objects, each assigned the "task of knowing" the transformation to be performed. (iii) A *heuristic optimizer* producing variations of the generated pipeline to improve performance by introducing perturbations and ad-hoc simplifications. (iv) Finally, a *query compiler* translating the logical graph structure into a *reasoning query plan*. Each placeholder generates a filter responsible for performing the transformations; each read-write dependency between rules induces a pipe. Figure 1 shows the Example 1.1 pipeline.
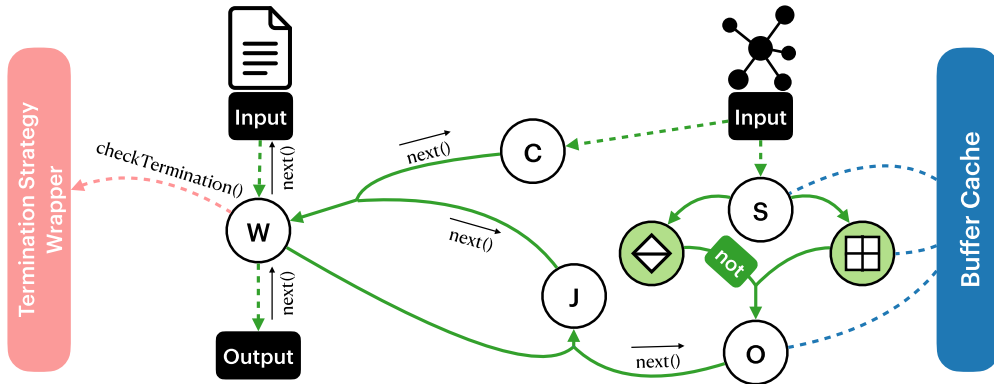
Fig. 1. The reasoning pipeline for Example 1.1. The atom *significantShare* is denoted by the filter S, *significantOwner* by O, *watchCompany* by W, *connected* by C, and J is an artificial filter to decompose, for simplicity, the ternary join of Rule 2 into binary joins.

At runtime

At runtime, a *pull-based* approach is used: sinks iteratively pull data using `next()` and `get()`, propagating calls through a filter chain to source filters. Source filters read from the initial data source via data adapters. Each filter applies transformations based on rule types (linear, joins, temporal operators, aggregations, etc.). In particular, a linear filter handles Vadalog linear rules, defined as rules having a single atom in the body.[1] As long as data is available in the cascade of invoked filters, `next()` succeeds. In Figure 1, for example, facts for the output filter W are generated both with the input data C and S and recursively through the filter J since the *watchCompany* rule is recursive.

Temporal challenges

We will now give a full overview of the system, pointing out how Temporal Vadalog provides support for tackling a number of time-relevant challenges.

- **Applying Temporal Operators.** Section 3.2 covers how the temporal operators (e.g., the ⬦ operator in Figure 1) are implemented in the pipeline.
- **Merging strategies.** The semantics for ⊟ requires the merging of adjacent and overlapping intervals to function correctly. How merging strategies work and are designed in Temporal Vadalog is covered in Section 3.3.
- **Temporal Joins and Stratified Negation**. Section 3.4 explains the implementation of joins in temporal reasoning (e.g. in the W filter), where time awareness, as well as support for stratified negation, is required.
- **Termination Strategies**. DatalogMTL can formulate programs with infinite models (Bellomarini et al. 2021b), by capturing domain events that repeat infinitely, such as weekdays. Section 3.5 outlines our strategy for program termination.

---

[1] Different definitions of 'linear rule' exist in the literature, which can lead to ambiguity when comparing works and perspectives. To avoid confusion, we will refer to our version as the "Vadalog linear rule."

- **Aggregate functions and numeric operations.** The Temporal Vadalog System provides standard scalar and temporal arithmetic operations, as well as support for aggregate functions in the form of time point or cross-time monotonic aggregations. This unique feature allows for a non-blocking implementation that works seamlessly with recursion. As far as we know, the Temporal Vadalog System is the only DatalogMTL reasoner capable of implementing aggregations. Their syntax and semantics are explained in our recent publication (Bellomarini et al. 2021a).
- **Temporal and non-Temporal Reasoning**. Since many existential extensions are undecidable (Lanzinger and Walega 2022), we consider temporal reasoning and existential reasoning as orthogonal fragments: in Datalog with existentials (Bellomarini et al. 2022) we forbid temporal operators and in DatalogMTL we forbid existentials. To support both modes within one program we added support for temporal wrapping and unwrapping of rules by allowing to switch between intervals as metadata and as atoms.
- **Time Series**. The handling of time series and the most common operators in this kind of analysis will be presented in Section 4.

3.2 Temporal operators

The implementation of the temporal operators introduced by DatalogMTL is designed in a two-phase procedure, first by applying logical optimization to reduce the number of required rules, and then by introducing multiple filter nodes in the reasoning pipeline.

Logical optimization

The application of linear temporal operators, that is, the operators for Vadalog linear rules, *box* and *diamond*, results in intervals computed through mathematical expressions between the interval of a fact, $\langle t_1, t_2 \rangle$, and that of the operator itself, $\varrho = \langle o_1, o_2 \rangle$, as schematized by Walega et al. (2019):

$$\langle t_1 + o_1, t_2 + o_2 \rangle \quad (\Diamond) \qquad \langle t_1 - o_2, t_2 - o_1 \rangle \quad (\oplus)$$
$$\langle t_1 + o_2, t_2 + o_1 \rangle \quad (\boxminus) \qquad \langle t_1 - o_1, t_2 - o_2 \rangle \quad (\boxplus)$$

with corresponding rules for determining the interval boundaries.[2]

Among these, $\Diamond$ offers one of the most accessible intuitions for understanding how an interval is derived: let us say that a shop has *recentlyOpened* if it had its inauguration in the past 12 days: $\Diamond_{[0,12]} inauguration(X) \rightarrow recentlyOpened(X)$. Assume that shop A has had its inauguration on the 5th and 6th days of the month: $D = \{inauguration(A)@[5,6]\}$. Then, A can be considered to have *recentlyOpened* between the 5th and the 18th: $recentlyOpened(A)@[5, 18]$, that is $[t_1 + o_1, t_2 + o_2]$.

As these expressions can be composed, one can rewrite a chain of Vadalog linear rules containing temporal operators[3] into a single rule with an equivalent temporal operator, as long as every intermediate step of the chain yields a valid (i.e. non-empty) interval.

---

[2] To be concise, we only see examples with closed intervals, where the application of temporal operators does not change them. The transformation tables can be found in Walega et al. (2019).
[3] The logical optimization does not consider $\mathcal{S}$ (*Since*) or $\mathcal{U}$ (*Until*) as these operators are not linear.

Since the derived interval can include both positive and negative endpoints, an equivalent operator cannot be chosen among the linear operators $\ominus$, $\oplus$, $\boxminus$, $\boxplus$, and instead is implemented as a generic operator $T\langle e_1, e_2 \rangle$ where $\langle e_1, e_2 \rangle$ is the desired interval transformation. That is, $T\langle e_1, e_2 \rangle$ applied to the interval $\langle t_1, t_2 \rangle$ of a fact gives $\langle t_1 + e_1, t_2 + e_2 \rangle$. Note that T is not constrained by the usual restrictions for intervals in linear operators, that is e_1 and e_2 are arbitrary for T. The linear operators $\ominus$, $\oplus$, $\boxminus$, $\boxplus$ can easily be seen as instances of the generic T operator: we have that $T\langle e_1, e_2 \rangle$ expresses any of the following: $\ominus_{\langle e_1, e_2 \rangle}, \boxminus_{\langle e_2, e_1 \rangle}, \oplus_{\langle -e_2, -e_1 \rangle}$, or $\boxplus_{\langle -e_1, -e_2 \rangle}$, or $\boxplus_{\langle -e_1, -e_2 \rangle}$.

Algorithm

Given a chain of Vadalog linear rules, we apply the mathematical expressions of each operator in order, starting from $[0, 0]$, the smallest valid interval for facts. If the resulting interval is empty – that is, the left endpoint is larger than the right endpoint, or, in case it is equal, the interval is open – or if we reach the end of the chain, we stop and merge the rules to an equivalent temporal operator applying the derived interval. This process is applied exhaustively until all rules are merged.

Example 3.1.
Consider the following combination of three temporal operators, where (Expression 3) can be reduced to $T[9, 9]$, while (Expression 4) can be reduced only partially.

$$\boxminus_{[5,10]}\ominus_{[2,5]}\oplus_{[1,3]}A \tag{3}$$

$$\ominus_{[2,5]} \boxminus_{[5,10]} \oplus_{[1,3]}A \tag{4}$$

For Expression 3, we first apply $\oplus_{[1,3]}$[4] to $[0, 0]$, resulting in $[-3, -1]$, then apply $\ominus_{[2,5]}$, resulting in the interval $[-1, 4]$. Then, we combine this interval with the remaining operator $\boxminus_{[5,10]}$, resulting in $[9, 9]$ and in the final expression $T[9, 9]A$. For Expression 4, we again apply $\oplus_{[1,3]}$ to $[0, 0]$, resulting in $[-3, -1]$, but then apply $\boxminus_{[5,10]}$ resulting in the interval $[7, 4]$. As no further optimization is possible, the final expression is $\ominus_{[2,5]}T[7, 4]A$.

Reasoning pipeline

A filter node is introduced for each operation executed in the pipeline. For the operators introduced by DatalogMTL, we consider the following:

- *TemporalNode.* This node computes the resulting interval of a fact at the application of a linear temporal operator ($\ominus$, $\oplus$, $\boxminus$, $\boxplus$, or the generic T operator introduced in *Logical Optimization*, following the DatalogMTL semantics as shown above).
- *MergeNode.* This node merges overlapping and adjacent intervals, required for the application of the box operator. Its functioning is addressed in Section 3.3.
- *ClosingNode.* This node computes the closure of the interval of a fact. It is required before the application of $\mathcal{S}$ and $\mathcal{U}$, as defined in (Walega et al. 2019): $\varrho_{\mathcal{S}} = ((\varrho_1^c \cap \varrho_2) + \varrho_3) \cap \varrho_1^c$, resp. $\varrho_{\mathcal{U}} = ((\varrho_1^c \cap \varrho_2) - \varrho_3) \cap \varrho_1^c$ where c denotes the closing operator and $+$ (resp. $-$) is the addition (subtraction) of intervals as defined for $\oplus$ ($\ominus$).

---

[4] Note that the operators are applied from inside to outside to the given fact.

- *TemporalJoinNode.* An extension of the join node in Vadalog incorporating restrictions on intervals of facts, required for $\mathcal{S}$ and $\mathcal{U}$ to take the semantics of the special join behavior of intervals into account. We discuss this node in detail in Section 3.4.

3.3 Merging strategies

This section discusses the implementation of the MergeNode, which provides us with two orthogonal optimization choices: (i) the placement strategy and (ii) the merge strategy.

Example 3.2.
A person X that owns an amount Z of shares of a company Y is an investor of the company (Rule 5), and a long-time investor of the company if it is an investor for at least 3 years with gaps of at most half a year (Rule 6).

$$shares(X, Y, Z) \rightarrow investor(X, Y) \tag{5}$$

$$\boxminus_{[0,3]} \diamondsuit_{[0,0.5]} investor(X, Y) \rightarrow longTimeInvestor(X, Y) \tag{6}$$

with the following database:

$$\mathcal{D} = \{ shares(A, B, 0.2)@[0.1, 0.5), shares(A, B, 0.2)@[0.4, 1.1),$$
$$shares(A, B, 0.3)@[1.5, 3.7), shares(A, B, 0.4)@[3.7, 4.2) \}$$

Placement strategy

The system maintains an ordered data structure of intervals for each fact. Depending on the program, there are multiple ways to merge facts to uphold correctness. Consider Example 3.2, where one can merge directly before the box operator (minimal merge), before the diamond operator (earliest merge), or continuously, that is always guaranteeing that all intervals per fact are merged (always merge) throughout the reasoning process. The placement options are described as follows:

- *Minimal Merge.* The planner inserts a merge operation before every box operator. This ensures correctness but does not provide further performance optimization.
- *Earliest Merge.* The planner inserts the merge in the earliest position it can be processed in the pipeline, so that the intervals of each fact are already merged when it reaches the box operator, while the operators coming before the latter also benefit from the reduced number of facts. If no merge operation is required (i.e. there is no box operator) the planner avoids merging.
- *Always Merge.* Merge operations are placed after the input filters and all filters that can generate facts with unmerged intervals. This can reduce the number of intermediary facts, providing a memory usage improvement, yet has a tradeoff with the time required for merging all existing facts.

In addition, for the *Minimal* and *Earliest Merge*, further optimization is obtained by providing the planner hints on where to place additional merge nodes, such as: after rules that share the same head; after a diamond operator produces many overlapping intervals; after the input or before the output to eliminate duplicates.
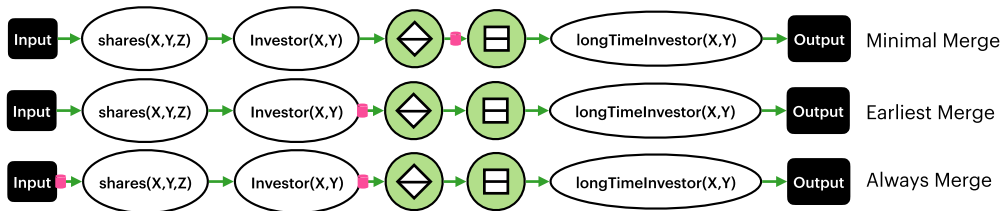
Fig. 2. Overview of interleaving strategies; merging positions are marked in fuchsia.

Algorithm 1 Streaming Strategy in the MergeNode

1: mergeStructure ← *createMergeStructure*()
2: **function** Next
3:      next ← *Linear.next*()
4:      **if** next **then**
5:          (changed, mergedEntry) ← mergeStructure.*add*(*Linear.get*())
6:          *set*(mergedEntry)
7:      **return** next

Figure 2 shows the merging strategies for Example 3.2. Temporal operators transform any set of intervals into a merged set: coalescing is always applied before the output.

Merge strategy

In software development, we face a typical tradeoff between streaming and blocking-based processing operations: the former typically aims for system responsiveness, limited memory footprint, and in-memory computation, while the latter aims for improving overall performance but requires more memory or materialization of intermediate results (Sciore 2020). In the case of the merging operator, we recognize a similar behavior. The merging operator is partially blocking: it can return a partial merge, but has to wait for all facts to be processed to return the final merged intervals. Two implementation options, *Streaming* and *Blocking*, were integrated into the system. Algorithms 1-2 present these strategies through the function next(), which returns a Boolean denoting whether a new fact has been retrieved (the caller will later retrieve the fact itself through a separate getter function get()). It makes use of *Linear.next()*, inherited from the linear filter scan, to check for new facts in the pipeline; thus it retrieves the facts, merges their intervals, and saves them into a *mergeStructure*, a hash map with the ground atoms as keys and the collections of intervals as values.[5] Every node (and hence, every MergeNode) handles only one atom. Updating *mergeStructure* via *add* also returns a Boolean *newChange* to note whether *mergeStructure* has been updated (i.e. it added new facts or new intervals not subsumed by already-present intervals.)

---

[5] Intervals are stored in a tree-like structure that merges those adjacent and overlapping automatically.

Algorithm 2 Blocking Strategy in the MergeNode

1: mergeStructure ← *createMergeStructure*()
2: currentPosition ← 0
3: **function** Next
4:    changed ← *false*
5:    **if** currentPosition ≥ mergeStructure.*length* **then**
6:      **while** *Linear*.*next*() **do**
7:        (newChange, mergedEntry) ← mergeStructure.*add*(*Linear*.*get*())
8:        changed ← changed or newChange
9:    **if** changed **then**
10:      currentPosition ← 0
11:    *set*(mergeStructure.*getAtPosition*(currentPosition)) ▷ set() automatically handles nulls
12:    currentPosition ← currentPosition + 1
13:    **return** currentPosition < mergeStructure.*length*

Streaming. (Algorithm 1) This operator pulls a fact from the previous node, merges the fact on-the-fly with the intermediate merging result, and forwards the merged fact without waiting until all incoming data has been processed.

Blocking. (Algorithm 2) This operator receives and merges all available facts before forwarding them: *currentPosition* keeps track of the facts already served downstream. Newly received facts[6] are only retrieved if the facts in *mergeStructure* have already been handled (Line 5); *currentPosition* is reset to 0 if *mergeStructure* is *changed*: as we do not know which facts have been changed,[7] all of them will have to be served again downstream (Lines 9-10). With *set()* (Line 11) we specify which fact will be retrieved with get() by the next() call from the filter below (an incremental value as long as the next() is called and no new facts are retrieved, 0 in case we have to start again because the structure changed). The calls are performed as long as there are new facts to pull from the filter above. By construction, as shown in Section 3.1, our termination strategies ensure termination also in the presence of cyclicity. Note that the possible merges of the facts have already been performed during the execution of the pipeline and no outstanding merges are left after termination.

Example 3.3.
Example 3.2 (continued) with the merge operator placed directly before the box. The Streaming Merge reads the first three entries, sufficient to apply $\boxminus_{[0,3]}$ to derive the intermediate result longTimeInvestor(A,B)@[3.1,4.2], before the final longTimeInvestor(A,B)@[3.1,4.7] is derived; the Blocking Merge strategy waits for all facts to be read then applies the box operator, deriving only fact longTimeInvestor(A,B)@[3.1,4.7].

---

[6] We have to be aware of changes in case the node is part of a cyclic structure.
[7] This behavior has been chosen as a trade off between space and frequency of occurrence.

3.4 Temporal joins

Example 3.4.
A person X that goes to a movie matinee M (between $[14, 16)$) gets a discounted ticket (Rule 7 and database). In the example, only person A will get a discount.

$$goesToTheMovies(X, M), matineeDiscount(M) \rightarrow discountedTicket(X) \qquad (7)$$

$$\mathcal{D} = \{goesToTheMovies(A, C)@[15, 17], goesToTheMovies(B, C)@[21, 23],$$
$$matineeDiscount(C)@[14, 16]\}$$

When joining two or more temporal predicates, we are interested in joining them not only based on their terms, but also on the temporal interval of each fact. We call this particular join a "temporal join."

The temporal join in Temporal Vadalog is an extension with intervals of the Vadalog slot machine join (Bellomarini et al. 2022), an enhanced version of the index nested loop join (Garcia-Molina et al. 2009) with the support of dynamic in-memory indexing: instead of having a pre-calculated index, it builds the index in-memory during the first full scan. Algorithm 3 shows the operation for joining two predicates ($n = 2$). For each predicate A_k with $0 \leq k < n$ to be joined, we first use the index to get the next scanned fact that matches the known terms used in the join, that is the *joinTerm* from the previous A_j with $0 \leq j < k$ (Line 8). If no further fact is found in the index, we do the full scan (Lines 10-14) until either a matching fact is found or the number of facts is exhausted. If no further fact is found (Line 15-22), we continue the scan with the next A_j if it exists and the current A_k is not negated. In case a fact has been found (whether from the index or the full scan), we update the valid interval of the joined fact depending on the join logic (Line 23): the difference for a negated literal, the intersection for a positive interval, or a mixture of interval operations and set operations for $\mathcal{U}$ and $\mathcal{S}$. Finally, we check whether the resulting interval is valid and if not, if A_k is negated, we continue the scan with the next A_j if it exists (Line 25-29), otherwise we continue with the loop; in case the interval is not empty and A_k is not negated, we return true as we have found a valid joined fact (Line 31-32); otherwise, we continue to retrieve the next "negated" fact.

As the temporal join supports negated atoms, it follows that Temporal Vadalog supports stratified negation in safe rules.

3.5 Termination strategy for the infinite chase of intervals

In previous work (Bellomarini et al. 2021b), we discussed the fragments of DatalogMTL that can generate infinite models that have finite representations in DatalogMTL^{FP}. In the following example, we depict one such case. This section explains how Temporal Vadalog handles these cases to guarantee termination.

Example 3.5.
A 30-day Job Report is an economic indicator, released once every 30 days, whose content can impact stock prices. Stock market analysts want to be aware of this indicator's releases to mark them as possible causes of stock price change over a certain threshold (K below).

Algorithm 3 Temporal Join between two predicates

Input: predicates A_0 and A_1 to be joined, with A_0 not negated
1:   $I_0 \leftarrow A_0.iterator()$
2:   $I_1 \leftarrow A_1.iterator()$
3:   $(a_0, \varrho_0) \leftarrow I_0.getNext()$
4:   **function** Next
5:     $interval \leftarrow \varrho_0$
6:     **while** true **do**
7:       $X \leftarrow a_0.joinTerm$
8:       $(a_1, \varrho_1) \leftarrow A_1.index.get(X)$
9:       **if** a_1 is null **then**   ▷ Continue full scan, if index miss
10:         **while** $I_1.next()$ **do**
11:           $(a_1, \varrho_1) \leftarrow I_1.getNext()$
12:           $A_1.index.put(a_1)$ ▷ Update the index map for A_1
13:           **if** $a_1.joinTerm == a_0.joinTerm$ **then**
14:             **break**   ▷ Exit the inner loop if matching fact is found
15:       **if** a_1 is null **then**   ▷ No further matching fact A_1 found
16:         **if** A_1 is negated **then**
17:           **return** true
18:         **if** $I_0.next()$ is false **then**
19:           **return** false
20:         $(a_0, \varrho_0) \leftarrow I_0.getNext()$ ▷ Repeat loop for next a_0
21:         $interval \leftarrow \varrho_0$
22:         **continue**
23:       $interval \leftarrow resultingIntervalFromJoinLogic(\varrho_1, interval)$
24:       **if** $interval$ is empty **then**
25:         **if** A_1 is negated **then**
26:           **if** $I_0.next()$ is false **then**
27:             **return** false
28:           $(a_0, \varrho_0) \leftarrow I_0.getNext()$ ▷ Repeat loop for next a_0
29:           $interval \leftarrow \varrho_0$
30:         **continue**
31:       **if** A_1 is not negated **then**
32:         **return** true

$$\diamondsuit_{[30,30]} \mathit{30DayJobReport} \rightarrow \mathit{30DayJobReport} \tag{8}$$

$$\mathit{StockPriceChange}(X, V), V > K \rightarrow \mathit{PriceEvent}(X) \tag{9}$$

$$\mathit{PriceEvent}(X), \diamondsuit_{[0,1]} \mathit{30DayJobReport} \rightarrow \mathit{PossibleCause}(X, \text{"JR"}) \tag{10}$$

$$D = \{\mathit{30DayJobReport}@[0, 0], \mathit{StockPriceChange}(A, K + 1)@[121, 121], ...\}$$

A DatalogMTL^{FP} (resp. BP) program can fall into one of three categories (Bellomarini et al. 2021b): (i) it is *harmless*, that is it satisfies the sufficient conditions for a finite model; (ii) it is either DatalogMTL$^{FP}_{\diamondsuit}$ *temporal linear* or DatalogMTL$^{FP}_{\boxminus}$ *union-free*, that is it satisfies the sufficient condition for a constant model under certain constraints; (iii) otherwise, it belongs to the DatalogMTL^{FP} category, a sufficient condition for a *periodic* model.

The Temporal Vadalog system guarantees termination by employing a two-phase approach, one at *compile time* and one at *runtime*.

Compile time

In this phase, the planner detects the fragment of DatalogMTL^{FP}:

(i)   The planner checks if the program has "harmful" temporal cycles using (Bellomarini et al., 2021b, Algorithm 1). If the program is *harmless*, it marks $modelKind = Finite$.

(ii) In presence of harmful temporal cycles, the algorithm checks if the program is *temporal linear*, that is, each rule has at most one body predicate mutually temporal recursive with the head in the dependency graph of Π. If the program is temporal linear and the operators allowed in temporal linear, defined over $[t1, t2]$ are such that $t1 \neq t2$, then the algorithm sets the $modelKind = Constant$.

(iii) If the program is not temporal linear, the algorithm checks if the program is *union-free*. A program is union-free if there are no rules of Π sharing the same head predicate. If the program is union-free and the box operators $\boxminus_{[t1,t2]}$ are such that $t1 \neq t2$, then the algorithm sets the $modelKind = Constant$.

(iv) If the program does not meet the conditions to be temporal linear or union-free, we assume it is in DatalogMTL^{FP} and set $modelKind = Periodic$.

At this time, the system computes the repeating pattern length $pLength$ in all non-finite cases, based on the pattern lengths combination of the different Strongly Connected Components of Π. The resulting facts are of the form $P(\tau)@\varrho$ and $\{P(\tau)@\langle o_1, o_2\rangle, n\}$, while 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 case of *periodic*, or $\langle o_1 + x * pLength, \infty\rangle$, in the case of *constant*. Functional components called *termination strategies* wrap all filters in the pipeline, to prevent the generation at runtime of facts leading to non-termination.

Runtime

In this phase, the system behavior depends on the detected fragment (the one associated with the set *modelKind*), denoting *fragment awareness*. If *modelKind* is *finite*, non-termination can only be caused by Datalog recursion, and standard termination strategies are employed. The reasoning produces facts of the form $P(\tau)@\varrho$. If the model is *constant* or *periodic*, the termination strategies recognize facts generated by the "non-finite" filters and mark the matching repeating patterns. In the case of a *constant* model, the base interval of ground atoms is converted into $\langle o_1 + x * pLength, \infty\rangle$, and the generation stops. If the model is *periodic*, the termination strategies related to the "non-finite" filters represent the numeric intervals with the equivalent of their pattern-based symbol, so that repeating sub-intervals are not generated, and termination is guaranteed.

Looking back at Example 3.5, at compile time the planner determines that $modelKind = Periodic$, and, from Rule 10, that $pLength = 1$. At runtime, after fact *30DayJobReport*@[60, 60] is generated, the termination strategy for *PossibleCause* infers that $n = 0$ and so that all facts generated from Rule 8 have form *30DayJobReport*@[$x \times$
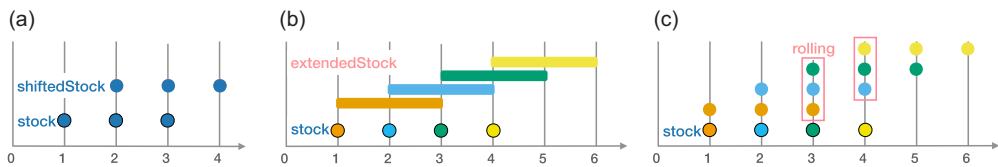
Fig. 3. Basic time series operators in Temporal Vadalog: a) Shifting by 1; b) Rolling operator with $n = 3$; c) Join of the extended intervals with the original ones.

$30, x \times 30 + 1]$ for $x \geq 0$, and their generation is put on hold. Applying the join for *PriceEvent*@$[121, 121]$ and the pattern generated from Rule 8-Rule 10, $x \in \mathbb{N}$ such that $[x \times 30, x \times 30 + 1] \cap [121, 121]$ is not an empty interval and corresponds to $x = 4$. We can conclude that *PossibleCause*$(A, \text{"JR"})$@$[121,121]$.

4 Time Series in Temporal Vadalog

Time series analysis is vital across many sectors, using historical data to identify trends, detect anomalies, predict events, and inform decisions to improve efficiency, reduce costs, and allocate resources. While many Time Series Database systems exist, considering time series analysis relevance in data science, we want to explore how to handle it with a more general-purpose system like Temporal Vadalog. allowing us to use common operators for time series in the broader scope of reasoning applied, for instance, to KGs, specifically exploiting its characteristics of explainability and context-awareness, especially important for domains dealing with sensitive data and particular needs.

In this section, we will present how the Temporal Vadalog System can be used to reason over time series, by showing how many of the core functions of time series databases are easily and immediately available by using DatalogMTL operators, monotonic temporal aggregations and arithmetic operations. Since we focus on regular time series, the examples in the following sections are intended to be used in discrete time (e.g. timestamps).

4.1 Basic operations

Temporal Vadalog natively handles the *sum* and other arithmetical operations over the same time intervals. This section covers other basic operations that deal with time series: shifting, rolling, and resampling. We will proceed informally by example, as the temporal operators' semantics has already been introduced.

Shifting

Shifting re-positions the time points of a time series by adding a delay, achieved through the box or diamond operator by stating a fact holds at t if it held at some past time point. The example adds a lag of one unit of time to the stock facts, where *stock*$(X, Value)$ represents any time series X with value *Value*. Figure 3a shows the shifting operation over a discrete timeline.

$$\diamondsuit_{[1,1]} stock(X, \, Value) \rightarrow shiftedStock(X, \, Value) \tag{11}$$

Rolling

Rolling defines a fixed-size window that slides through a time series one data point at a time, allowing for the computation of statistics over the data points within the window. A rolling window of size n is achieved by extending each data point's interval by $[0, n)$ with the diamond operator (Figure 3b). The time series is later joined with the original one, applying the full window's data points to each specific interval for further operations (Figure 3c where at $t = 3, 4$ we have the $n = 3$ rolling windows.)

$$\diamondsuit_{[0,n)} stock(X, \, Value) \rightarrow extended(X, \, Value) \tag{12}$$

$$stock(X, \, Value), \, extended(X, \, Roll) \rightarrow rolling(X, \, Roll) \tag{13}$$

Resampling

The resampling of a time series changes its time resolution. Working with different time resolutions (e.g. one monthly series and one daily) may require this change in both directions: *downsampling*, lowering the frequency of the data (e.g. daily to monthly), and *upsampling*, increasing the frequency (e.g. monthly to daily).

Downsampling. In Temporal Vadalog we can achieve *downsampling* by using the aggregation operator $\triangle_{unit}$, where *unit* is the frequency we want to transform our time series into. This operator was introduced in a previous work (Bellomarini et al. 2021a).

$$\triangle_{month} dailyStock(X, \, Value) \rightarrow monthlyStocks(X, \, Value) \tag{14}$$

The set of facts that were valid daily will be now valid over the entire month. To obtain the final value for the monthly data point, we can use arbitrary aggregations, depending on the domain (e.g. arithmetic mean, minimum, maximum, etc.)

Upsampling. We can convert a time series to a higher frequency by using the temporal join with a series with a higher frequency that we can create on the spot with a diamond operator. In the following example, we will convert a *weeklyStock*(Company,Value,WeekStart), where *WeekStart* is expressed in a day unit, into a daily time series.

$$\diamondsuit_{[1,1]} dailySeries(X), Y = X + 1 \rightarrow dailySeries(Y) \tag{15}$$

$$\diamondsuit_{[0,7)} weeklyStock(C, V, W) \rightarrow nextWeek(C, V, W) \tag{16}$$

$$dailySeries(Z), \, weeklyStock(C, V_1, W_1),$$
$$nextWeek(C, V_2, W_2), W = W_1 + \tfrac{(V - V_1)(W_2 - W_1)}{(V_2 - V_1)} \rightarrow upsampling(C, W) \tag{17}$$

Note that Rule 15 generates an infinite chase of intervals, which requires us to employ a mechanism for terminating the generations like the strategies discussed in Section 3.5.
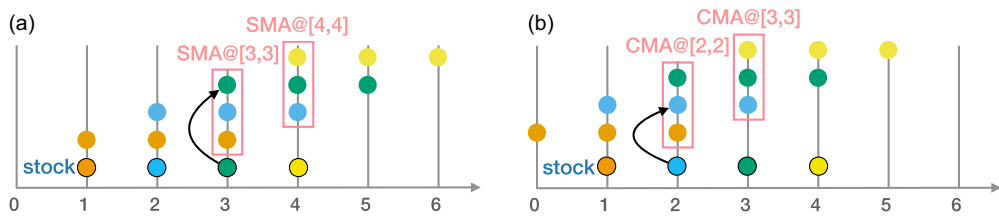
Fig. 4. From simple moving average (a) to centered moving average (b), with $n = 3$.

4.2 Moving averages

Moving averages use the rolling window described above to calculate different types of averages, typically to smooth out fluctuations and noise.

Simple moving average (SMA)

In this moving average, each data point contained in the window has the same weight in the calculation of the average. Hence, we compute first the rolling window and then derive the arithmetic mean of the data points (Figure 4a).

$$rolling(X, Y), Avg = avg(Y) \rightarrow sma(X, Avg) \tag{18}$$

Exponential moving average (EMA)

With the EMA operator we give more importance to the more recent data points. The first value is calculated through the simple moving average. We omit this from the example for conciseness. The following values are calculated as a sum (Rule 21) of the current data point multiplied by the parameter k (where $k = 2/(1 + n)$, n being the size of the window) (Rule 20) and, using recursion, the previous moving average multiplied by $1 - k$ (Rule 19).

$$\diamondsuit_{[1,1]} ema(X), Y = X * (1 - k) \rightarrow emaInput(Y) \tag{19}$$

$$stock(X, V), Y = X * k \rightarrow emaInput(Y) \tag{20}$$

$$emaInput(X), Y = sum(X) \rightarrow ema(Y) \tag{21}$$

Centered moving average (CMA)

In the SMA operator, a window of n data points will yield a value at the right extremity of that window; in the Centered Moving Average (CMA), instead, the data point of reference for the moving average is the data point at the center of the window. For any odd n window, this is obtained by calculating the SMA and then shifting it by $n/2$, as shown by Figure 4a-b.

$$\diamondsuit_{[0,n/2)} sma(X, Y) \rightarrow cma(X, Y) \tag{22}$$

4.3 Stock and flow

Performing mutual conversion between stock and flow is a helpful feature in multiple contexts. Transformation operators, in continuous time, can be computed through derivatives and integrals, while in discrete time one usually considers the difference between two time points. We focus on the discrete version, as the discrete (integer) timelines are the usual implementation choice in time series databases (Dyreson 2018).

Stock to flow (discrete)

This operator calculates the flow as the difference between the stock at t and $t-1$. To do so in Temporal Vadalog, we shift the time series by $[1, 1]$ and compute the difference between the values from the original and shifted time series.

$$\diamondsuit_{[1,1]} stock(X, V) \rightarrow shiftedStock(X, V) \tag{23}$$

$$stock(X, V1), shiftedStock(X, V2), V = V1 - V2 \rightarrow stockToFlow(X, V) \tag{24}$$

Flow to stock (discrete)

This operator, on the other hand, calculates the stock as the cumulative sum of flows from the beginning of the time series up to the current data point. To do so in Temporal Vadalog, we extend the flow time series to the interval $[0, T]$, where T is the last interval, and then we compute the sum over each data point.

$$\diamondsuit_{[0,T)} flow(X, V) \rightarrow rolledFlow(X, V) \tag{25}$$

$$rolledFlow(X, V1), V = sum(V1) \rightarrow stock(X, V) \tag{26}$$

4.4 Seasonal decomposition

Seasonal decomposition separates a time series into three different components: trend, seasonality, and residual. Regular, repeating events such as holidays and seasonal shifts impact variables similarly from one period (e.g. a year) to the next. For this reason, extracting the seasonality from the time series allows having a better understanding of the data. In this section, we compute seasonal decomposition with the additive model.

Trend

The trend component is calculated as the Centered Moving Average (CMA) of the time series, as explained in Section 4.2, abbreviated in the following as a call to *cma* with T the length of the period (e.g. 12 months).

$$stock(X, Stock), Value = cma(Stock, T) \rightarrow trend(X, Value) \tag{27}$$

Seasonal component

The seasonal component is extracted by calculating the per-period averages over the difference between the time series and the *trend* component, which we call *detrend*. In

the following example, we calculate the seasonal component over the period N (e.g. 10 years), with T representing the length of the period.

$$\diamondsuit_{[T,T]} detrend(X, D), Sum = sum(D), Value = Sum/N \rightarrow seasonal(X, Value) \quad (28)$$

Residual

Finally, the residual is the time series when the seasonal component and the trend are taken out.

$$detrend(X, D), seasonal(X, S), Value = D - S \rightarrow residual(X, Value) \quad (29)$$

This concludes our section on time series operations in Temporal Vadalog.

5 Experiments

For the experimental evaluation of our system, we conducted several performance tests. Previously (Bellomarini et al. 2022), we presented the Company Ownership experiments, a set of five scenarios about ownership changes in companies, whose historical information was stored in Knowledge Graphs of increasing sizes. In this paper, we present new results of the same scenarios, with the optimized reasoning capability of the current Temporal Vadalog and more dynamic datasets (Section 5.1), as well as new results from the experiments created with iTemporal (Bellomarini et al. 2022), an extensible generator of temporal benchmarks, in Section 5.4. We also show the results of new experiments: We replicated the experiments from the MeTeoR paper (Wang et al. 2022), the Lehigh University Benchmark ones in Section 5.2 and the Meteorological ones in Section 5.3. We also evaluate the performance of reasoning over time series in Section 5.5.

Setup

We run the experiments in a memory-optimized virtual machine with 16 cores and 256 GB RAM on an Intel Xeon architecture. Each experiment was run 3 times, and the results shown are the arithmetic mean of the elapsed time of each set of runs. The time out is set at 4,000 s (∼1 hour and 7 minutes).

Datasets

Our datasets comprise *real-world*, *realistic* and *synthetic* data. Our **Company ownership** experiments have been run over a real-world dataset (RW1721) extracted from the Italian Companies KG (Bellomarini et al. 2020), in a slice comprising the (continuous) ownership edges from 2017-21. The realistic datasets (MN7-MN28) represent company ownership graphs from 700 thousand to 2.8 million nodes generated in the likeness of a real-world dynamic structure, with their ownership changes over 5 intervals and a high change rate, to test performance. The datasets for the **Lehigh University Benchmark** are as described in the MeTeoR paper (Wang et al. 2022); the **Meteorological** experiments employ 3 datasets extracted from the Maurer et al. (2002) dataset at 5/50/500

stations. For the **iTemporal** experiments, the synthetic datasets (S1-S10M) are generated with a random distribution over a given domain and include from 1K to 10 M facts. The **Time Series** experiments were run over the NASDAQ Composite Index (NASDAQ OMX Group 2023), a daily time series from February 1971 to March 2023 comprising 13596 records. From this dataset, we extracted the last 1% (NQ1), 10% (NQ10), and 50% (NQ50) so to have 4 total data sizes. Details about the datasets employed in the experiments can be found in the supplementary material.

5.1 Company ownership experiments

Similarly to Example 1.1, we ran experiments on company ownership changes over time.

Scenarios

The experiment scenarios contain different elements: 1) *Temporal*: diamond operator, recursion, and constraints on variables; 2) *Negation*: recursion and stratified negation; 3) *Aggregation*, temporal aggregations; 4) *Diamond*: diamond operator and recursion; 5) *Box*: box operator and recursion. We ran each scenario on MN7-28 and RW1721; *Box* and *Diamond* were also run on MeTeoR 1.0.15 for comparison, as they do not include features not supported by MeTeoR.[8] All scenarios were run with the *always merge* strategy; *Box* was tested also in the *minimal* and *earliest merge*.

Discussion

The performance of *Temporal*, *Negation* and *Aggregation* are shown in Figure 5a. They all show good scalability, with the elapsed time increasing linearly over the dataset sizes. *Temporal*, the most complex scenario, is the most expensive as well but with just over 182 secs for the biggest datasets, while *Aggregation* and *Negation* perform two times faster at 24-99 secs. Figure 5b shows the results for *Box* and *Diamond* in both Vadalog and MeTeoR.[9] Vadalog performs in a similar linear-increase fashion to the other scenarios, about 100 times faster than MeTeoR, which exceeds the 4,000 secs threshold in the MN20-28 datasets in both scenarios. Merge strategy-wise, we see that the *always merge (am) Box* is the most expensive, running at 180 secs for MN28, while *earliest merge (em) Box* is on average 8% faster, and similar to those of *Diamond* at around 41-170 secs. The nature of the datasets, having many adjacent facts, explains it: by merging strategically, fewer facts are sent down the pipeline, and fewer operations are needed while merging all the facts at one time may be superfluous. Figure 5c shows the performance on the real-world dataset. MeTeoR results are not shown as they all exceed the timeout. In Vadalog, *Negation* and *Temporal* are the fastest at 102 secs, while *Aggregation* is the most expensive at 183 secs. In MN7-28, *Aggregation* performed well compared to *Temporal* and *Negation*, but in RW1721, it is the worst due to having 12 times more intervals, decreasing performance. As for the *merge strategies*, RW1721 shows similar performance to MN7-28, with *Box em*

---

[8] Due to an incompatibility between data and the *seminaive* evaluation in MeTeoR, which let the execution run indefinitely, we had to run the MeTeoR experiments in *naive* mode.

[9] Due to the difference in performance, the plot for MeTeoR is shown separately within the figure. Appendix, Figures 4-5 show a larger rendition of these curves.
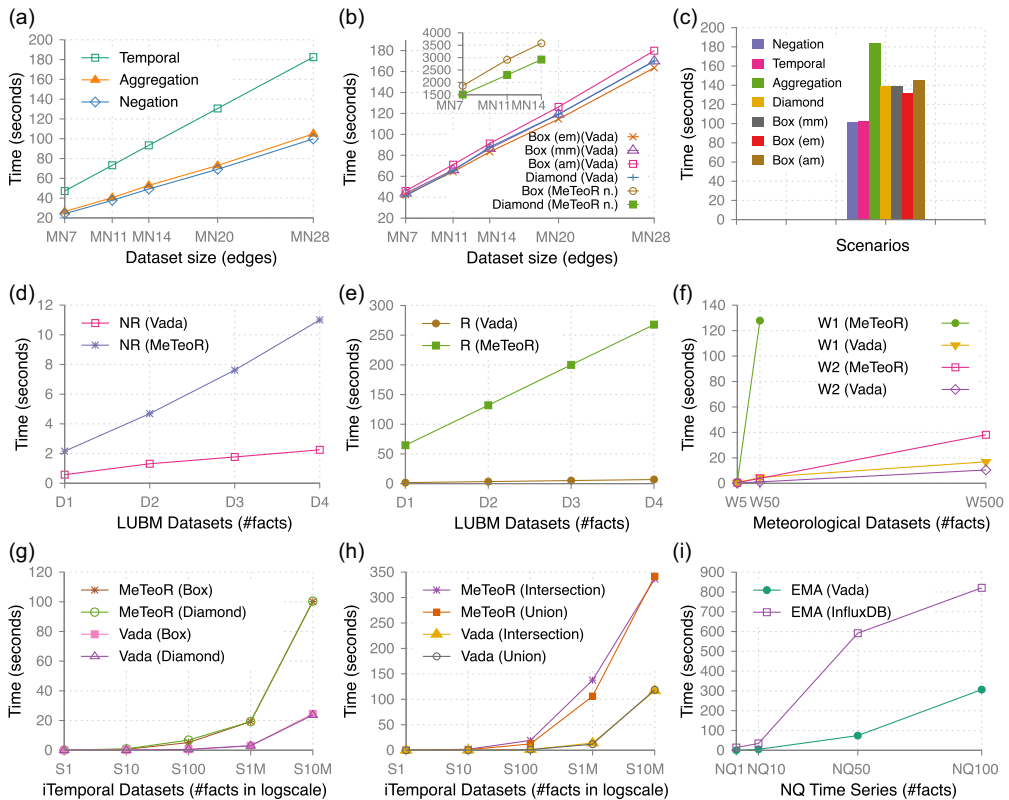
Fig. 5. (a) *Temporal*, *Aggregation* and *Negation* on MN7-MN28 over time; (b) *Box* and *Diamond* in Temporal Vadalog and MeTeoR; (c) RW dataset in all scenarios; (d) LUBM Non-Recursive experiment; (e) LUBM Recursive experiment; (f) Meteorological experiments; (g) *iTemporal* Box and Diamond; (h) *iTemporal* Union and Intersection; (i) Time Series.

the best performer at 131 secs and *Box am* the worst at 145 secs. The full result tables and larger versions of all graphs can be found in the supplementary material.

5.2 Lehigh university benchmark experiments (LUBM)

The LUBM benchmark for DatalogMTL (Wang et al. 2022) in a modified form Walega et al. (2023) measures the efficiency of forward-propagating rules in MeTeoR in the interval $[0, 300]$ for four different dataset sizes (5, 10, 15, and 20 million facts) and two different programs (one recursive and one non-recursive program).

Discussion

We used the published program and dataset from Walega et al. (2023) and repeated the experiments for MeTeoR (*seminaive* mode) and our system. The results are presented in Figures 5d-e and show an overhead for MeTeoR of a factor of at least 3 for the non-recursive program and of a factor of at least 30 for the recursive ones (R). The

main factor for better performance in Vadalog is the join, which in recursive programs especially shows the benefit of indexing already-seen atoms.

5.3 Meteorological experiments

We adapted the rules of the experiments on meteorological data (Maurer et al. 2002) in Wang et al. (2022). The programs W1 and W2 are run over a filtered version of this dataset: W1 regarding high temperatures, and W2 regarding heavy winds. For this reason, we have to consider that while the original dataset is the same, W1 works with approximately 10 times the facts of W2. MeTeoR runs in *seminaive* mode.

Discussion

The results are shown in Figure 5f. Both W1 and W2 perform well in Vadalog, with Vadalog being 28 times faster than MeTeoR for W1 at the W50 size (MeTeoR exceeds the time out for W500), and around 4 times faster for W2.

5.4 iTemporal experiments

Considering the previous experiments, we observe a substantial speedup of the Temporal Vadalog System with respect to MeTeoR (*seminaive* mode). To confirm these measurements, we used *iTemporal* (Bellomarini et al. 2022), a generator for DatalogMTL, to generate specific experiments with different data sizes targeting the main temporal operations (i.e., the application of temporal operators, joins, and unions).

Discussion

In Figures 5g-h we present the results, which show that our system outperforms MeTeoR with a factor of 3 to 4 (depending on the benchmark) for $10M$ facts and reinforces the observations of the previous benchmarks (e.g. it scales linearly).

5.5 Time series experiments

We executed the computation of the *exponential moving average* (EMA) on Vadalog and the Time Series Database InfluxDB (Version 2.6.1) over the NASDAQ Composite Index (NASDAQ OMX Group 2023). InfluxDB is a NoSQL time series database with a distinct data model optimized for time series operations. (InfluxDB 2023) For a fair comparison with the streaming-based approach of Vadalog, we compare the total time of loading and querying the data (end-to-end process).

Discussion

Results are shown in Figure 5i. Our system outperforms InfluxDB by 2.5x for the largest dataset, considering the end-to-end measurement. We note that the experiment's aim is not to compare against the performance of InfluxDB, but to show that our system can achieve reasonable reasoning times in an end-to-end manner.

6 Related Work

The 1980s see the first proposals for a temporal version of Datalog with Datalog$_{1S}$ (Chomicki 1990; Chomicki and Imielinski 1988, 1989), using successor functions, and Datalog extensions with temporal logic operators from Linear Temporal Logic with Templog (Abadi and Manna 1989), both with further work from Baudinet (1992, 1993, 1995) and Computation Tree Logic with Datalog LITE (Gottlob et al. 2002). Further work on constraint databases include: Brodsky et al. (1997); Maher and Srivastava (1996); Revesz (2002). Interest in the application of logic programming over the dense timeline focused on Metric Temporal Logic (Alur and Henzinger 1993; Koymans 1990) in the work of Brzoska (1995, 1998). More recently, MTL has been considered as a formalism that can provide an expressive temporal extension to Datalog with DatalogMTL in the work of Brandt et al. (2017, 2018), who also presented a practical implementation through SQL rewriting, although non-recursive. Theory for continuous semantics DatalogMTL (Tena Cucala et al. 2021; Walega et al. 2019; Walega et al. 2020a; Walega et al. 2020b) as well as for pointwise (Kikot et al. 2018) has been studied extensively in recent years, and most recently with MeTeoR (Wang et al. 2022; Walega 2023), a reasoner employing a combination of materialization and automata-based reasoning, and that we have referred for our comparative experiments.

Another comparable system, the Dyna language solver, proposed by Vieira et al. (2017) employs fixed point evaluation but computation may fail to terminate, while Vadalog employs the isomorphic chase with termination strategies, hence a program will always terminate; the Dyna solver uses forward or backward chaining depending on the strategy chosen by its reinforcement learning optimizer, whereas Vadalog uses a pull-based approach for evaluation, more efficient in resource usage for query answering.

The non-temporal version of the Vadalog System (Bellomarini et al. 2022) has been proposed as a system to reason over KGs (Bellomarini et al. 2020) with rules expressed in Warded Datalog$^{\pm}$ (Bellomarini et al. 2018). Other reasoners of similar expressive power include PDQ (Benedikt et al. 2015), Llunatic (Geerts et al. 2013), Graal (Baget et al. 2015), DLV (Leone et al. 2006), and RDFox (Nenov et al. 2015).

7 Conclusion

In this paper, we introduced a new framework and architecture for reasoning with DatalogMTL, highlighting its efficiency and ability to handle complex tasks, like the reasoning over time series, through multiple performance evaluations. We exceeded the capabilities of existing DatalogMTL reasoning tools. Moving forward, we aim to enhance the reasoner and explore other fragments of DatalogMTL.

Acknowledgements

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068425000018.

References

ABADI, M. and MANNA, Z. 1989. Temporal Logic Programming. *Journal of Symbolic Computation* 8, 3, 277–295.

ALUR, R. and HENZINGER, T. A. 1993. Real-Time Logics: Complexity and Expressiveness. *Information and Computation* 104, 1, 35–77.

BAGET, J., LECLÈRE, M., MUGNIER, M., ROCHER, S. and SIPIETER, C. (2015) Graal: A Toolkit for Query Answering with Existential Rules, *RuleML*, 9202, pp. 328–344.

BALDAZZI, T., BELLOMARINI, L., FAVORITO, M. and SALLINGER, E. (2022) On the Relationship between Shy and Warded Datalog+/−, *KR*.

BAUDINET, M. (1992) A Simple Proof of the Completeness of Ternporal Logic Programming, *Intensional Logics for Programming*. Oxford University Press.

BAUDINET, M. 1995. On the Expressiveness of Temporal Logic Programming. *Ideology & Consciousness* 117, 2, 157–180.

BAUDINET, M., CHOMICKI, J. and WOLPER, P. (1993) Temporal Deductive Databases. *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, pp. 294–320.

BELLOMARINI, L., BENEDETTI, M., CERI, S., GENTILI, A., LAURENDI, R., MAGNANIMI, D., NISSL, M. and SALLINGER, E. (2020) Reasoning on Company Takeovers during the COVID-19 Crisis with Knowledge Graphs, *RuleML+RR (Suppl.)*, 2644, pp. 145–156.

BELLOMARINI, L., BENEDETTO, D., GOTTLOB, G. and SALLINGER, E. 2022. Vadalog: A Modern Architecture for Automated Reasoning with Large Knowledge Graphs. *Information Systems* 105, 101528.

BELLOMARINI, L., BLASI, L., NISSL, M. and SALLINGER, E. (2022) The Temporal Vadalog System, *RuleML+RR*, 13752, Springer, pp. 130–145.

BELLOMARINI, L., FAKHOURY, D., GOTTLOB, G. and SALLINGER, E. (2019) Knowledge Graphs and Enterprise AI: The Promise of an Enabling Technology, *ICDE. IEEE*, 26–37.

BELLOMARINI, L., NISSL, M. and SALLINGER, E. 2021a. Monotonic Aggregation for Temporal Datalog. (Proceedings of the 15th International Rule Challenge, 2956.

BELLOMARINI, L., NISSL, M. and SALLINGER, E. 2021b. Query Evaluation in DatalogMTL - Taming Infinite Query Results. CoRR abs/2109.10691.

BELLOMARINI, L., NISSL, M. and SALLINGER, E. (2022) iTemporal: An Extensible Generator of Temporal Benchmarks, *ICDE. IEEE*, 2021–2033.

BELLOMARINI, L., SALLINGER, E. and GOTTLOB, G. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *Proceedings of the VLDB Endowment* 11, 9, 975–987.

BELLOMARINI, L., SALLINGER, E. and VAHDATI, S. (2020) Knowledge Graphs: The Layered Perspective, *Knowledge Graphs and Big Data Processing*, 12072, Springer, pp. 20–34.

BENEDIKT, M., LEBLAY, J. and TSAMOURA, E. 2015. Querying with Access Patterns and Integrity Constraints. *Proceedings of the VLDB Endowment* 8, 6, 690–701.

BRANDT, S., KALAYCI, E. G., KONTCHAKOV, R., RYZHIKOV, V., XIAO, G. and ZAKHARYASCHEV, M. (2017) Ontology-Based Data Access with a Horn Fragment of Metric Temporal Logic, *AAAI*. AAAI Press, 1070–1076.

BRANDT, S., KALAYCI, E. G., RYZHIKOV, V., XIAO, G. and ZAKHARYASCHEV, M. 2018. Querying Log Data with Metric Temporal Logic. *Journal of Artificial Intelligence Research* 62, 829–877.

BRODSKY, A., JAFFAR, J. and MAHER, M. J. 1997. Toward Practical Query Evaluation for Constraint Databases. *Constraints An International Journal*, 3, 4, 279–304.

BRZOSKA, C. (1995) Temporal Logic Programming in Dense Time. In LOGIC PROGRAMMING, J. W. Lloyd, eds. MIT Press, pp. 303–317.

BRZOSKA, C. 1998. Programming in Metric Temporal Logic. *Theoretical Computer Science* 202, 1-2, 55–125.

BUSCHMANN, F., HENNEY, K. and SCHMIDT, D. C. 2007. *Pattern-Oriented Software Architecture*. 4th ed. Wiley Publishing, Hoboken, NJ.

CALÌ, A., GOTTLOB, G. and LUKASIEWICZ, T. 2012. A General Datalog-Based Framework for Tractable Query Answering Over Ontologies. *Journal of Web Semantics* 14, 57–83.

CERI, S., GOTTLOB, G. and TANCA, L. 1989. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering* 1, 1, 146–166.

CHOMICKI, J. (1990) Polynomial Time Query Processing in Temporal Deductive Databases, *PODS*. New York, NY, USA, pp. 379–391.

CHOMICKI, J. and IMIELINSKI, T. (1988) Temporal Deductive Databases and Infinite Objects, *PODS*, pp. 61–73.

CHOMICKI, J. and IMIELINSKI, T. (1989) Relational Specifications of Infinite Query Answers, *SIGMOD*. ACM Press, pp. 174–183.

DYRESON, C. E. (2018) Chronon, *Encyclopedia of Database Systems*. 2nd ed. Springer.

GARCIA-MOLINA, H., ULLMAN, J. D. and WIDOM, J. 2009. *Database Systems - the Complete Book (2. ed.)*. Pearson Prentice Hall, Upper Saddle River, NJ.

GEERTS, F., MECCA, G., PAPOTTI, P. and SANTORO, D. 2013. The LLUNATIC Data-Cleaning Framework. *Proceedings of the VLDB Endowment* 6, 9, 625–636.

GELDER, A. V., ROSS, K. A. and SCHLIPF, J. S. 1991. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM* 38, 3, 620–650.

GOTTLOB, G. (2022) Adventures with Datalog: Walking the Thin Line between Theory and Practice, *AI*IA*. *Lecture Notes in Computer Science*, 13796, Springer, pp. 489–500.

GOTTLOB, G., GRÄDEL, E. and VEITH, H. 2002. Datalog LITE: A Deductive Query Language with Linear Time Model Checking. *ACM Transactions on Computational Logic* 3, 1, 42–79.

GRAEFE, G. and MCKENNA, W. J. (1993) The Volcano Optimizer Generator: Extensibility and Efficient Search, *ICDE*, pp. 209–218.

InfluxDB (2023). InfluxDB Key Concepts. Available at http://tinyurl.com/influxdbkc.

KIKOT, S., RYZHIKOV, V., WALEGA, P. A. and ZAKHARYASCHEV, M. (2018) On the Data Complexity of Ontology-Mediated Queries with MTL Operators Over Timed Words, *DL*, 2211

KOYMANS, R. 1990. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems* 2, 4, 255–299.

LANZINGER, M. and WALEGA, P. A. (2022) Datalog with Existential Quantifiers and Temporal Operators (Extended Abstract), *Datalog 2.0*, 3203, pp. 139–144.

LEONE, N., ALLOCCA, C., ALVIANO, M., CALIMERI, F., CIVILI, C., COSTABILE, R., FIORENTINO, A., FUSCÀ, D., GERMANO, S., LABOCCETTA, G., CUTERI, B., MANNA, M., PERRI, S., REALE, K., RICCA, F., VELTRI, P. and ZANGARI, J. (2019) Enhancing DLV for Large-Scale Reasoning, *LPNMR*, 11481, pp. 312–325.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S. and SCARCELLO, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic* 7, 3, 499–562.

LIN, F., FU, C., HE, Y., XIONG, W. and LI, F. 2022. ReCF: Exploiting Response Reasoning for Correlation Filters in Real-Time UAV Tracking. *IEEE Transactions on Intelligent Transportation Systems* 23, 8, 10469–10480.

MA, M., LEE, K., MAI, Y., GILMAN, C., LIU, Z., ZHANG, M., LI, M., REDFERN, A., MULLANEY, T., PRENTICE, T., MCDONAGH, P., PAN, Q., CHEN, R., SCHADT, E. and WANG, X. 2021. Extracting Longitudinal Anticancer Treatments at Scale Using Deep Natural Language Processing and Temporal Reasoning. *Journal of Clinical Oncology*, 39, 15 suppl, e18747–e18747.

MAHER, M. J. and SRIVASTAVA, D. (1996) Chasing Constrained Tuple-Generating Dependencies. In HULL R., Eds. *PODS*, 128–138.

MAIER, D., MENDELZON, A. O. and SAGIV, Y. 1979. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems* 4, 4, 455–469.

MAURER, E. P., WOOD, A. W., ADAM, J. C., LETTENMAIER, D. P. and NIJSSEN, B. (2002). A Long-Term Hydrologically Based Dataset of Land Surface Fluxes and States for the Conterminous United States, *J. Climate*, **15**, 3237–3251.

Nasdaq omx Group (2023). NASDAQ Composite Index, http://tinyurl.com/frednq, FRED, Federal Reserve Bank of St. Louis, Accessed: 2023-03-22.

NENOV, Y., PIRO, R., MOTIK, B., HORROCKS, I., WU, Z. and BANERJEE, J. (2015) RDFox: A Highly-Scalable RDF Store, *ISWC 2).*, 9367, pp. 3–20.

REVESZ, P. Z. 2002. *Introduction to Constraint Databases. Texts in Computer Science.* Springer.

SCIORE, E. 2020. *Database Design and Implementation.* Second ed. Springer, Cham, New York, NY.

TENA CUCALA, D. J., WALEGA, P. A., CUENCA GRAU, B. and KOSTYLEV, E. V. (2021) Stratified Negation in Datalog with Metric Temporal Operators, *AAAI*, pp. 6488–6495.

VIEIRA, T., FRANCIS-LANDAU, M., FILARDO, N. W., KHORASANI, F. and EISNER, J. (2017) Dyna: Toward a Self-Optimizing Declarative Language for Machine Learning Applications, MAPL @ PLDI. 8-17.

WALEGA, P. A. 2023. Computational Complexity of Hybrid Interval Temporal Logics. *Annals of Pure and Applied Logic* 174, 1, 103165.

WALEGA, P. A., CUENCA GRAU, B., KAMINSKI, M. and KOSTYLEV, E. V. (2019) DatalogMTL Computational Complexity and Expressive Power, *IJCAI*, pp. 1886–1892.

WALEGA, P. A., CUENCA GRAU, B., KAMINSKI, M. and KOSTYLEV, E. V. (2020a) DatalogMTL Over the Integer Timeline, *KR*, pp. 768–777.

WALEGA, P. A., CUENCA GRAU, B., KAMINSKI, M. and KOSTYLEV, E. V. (2020b) Tractable Fragments of Datalog with Metric Temporal Operators, *IJCAI*, 1919-1925.

WALEGA, P. A., KAMINSKI, M. and CUENCA GRAU, B. (2019) Reasoning Over Streaming Data in Metric Temporal Datalog, *AAAI*, pp. 3092–3099.

WALEGA, P., KAMINSKI, M., WANG, D. and GRAU, B. C. 2023. Stream Reasoning with DatalogMTL. *Journal of Web Semantics* 76, 100776.

WANG, D., HU, P., WALEGA, P. A. and GRAU, B. C. (2022) MeTeoR: Practical Reasoning in Datalog with Metric Temporal Operators, *AAAI*, pp. 5906–5913.