Monotonic Aggregation for Temporal Datalog

Luigi Bellomarini[1], Markus Nissl[2], and Emanuel Sallinger[2,3]

[1] Banca d'Italia, Italy
[2] TU Wien, Austria
[3] University of Oxford, UK

Abstract. Understanding time-based effects has become an important aspect for the analysis of Knowledge Graphs (KGs). We have seen this in different areas such as IoT or economics. Scalable solutions for using Datalog-based KGs with time are in their infancy and the usage together with aggregation has not been considered so far. Yet, one needs both aggregation and time-based analysis when analysing KGs such as those of economic phenomena. In this paper, we analyze monotonic aggregation over DatalogMTL, establishing the first work that covers full recursion like in Datalog, aggregation, and temporal reasoning.

Keywords: Knowledge Graphs · Datalog · Temporal Reasoning.

1 Introduction

Understanding time-based effects has become a central aspect of *reasoning* on *Knowledge Graphs* (KGs), particularly in specific but prominent application settings and business domains. They include: IoT, where context awareness requires aggregating temporal data from continuous (and heterogeneous) sources [4]; declarative business process management, where activities and tasks need careful scheduling and prioritization [21]; state-of-the-art security information and event management systems (SIEM) with time-based alert and log events [22]. More traditionally, a standard application for time-based reasoning lies in the analysis of economic phenomena, where time is a natural dimension of analysis (e.g., for time series). It is our experience that reasoning on KGs is being increasingly applied to economic settings, often characterized by a complex network of intertwined entities. It has been shown that a Datalog-based KG 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 [3], from which a number of paradigmatic use cases of temporal reasoning and aggregations emerge:

- **UC1: Revenue Calculation**. Shareholder are interested in the revenue of a company per week/month/year/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 snapshot provides deeper insights on the takeover determinants and improves accuracy in prediction.
– **UC4: Change of Control**. Analysts wish to analyse ownership structures over time, e.g., monotonically increasing or decreasing shareholding.

These four use cases highlight different aggregation scenarios. While UC1 requires fixed intervals (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). UC3 requires to aggregate temporal information recursively over structures (e.g., over paths of arbitrary length). We call the kind of aggregation used in the previous use cases *time-point aggregation* as aggregation is applied on single time points, or time points in an interval. 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 order to support such use cases in Datalog-based reasoning, a **temporal extension of Datalog for aggregation over time** is required. MTL (Metric Temporal Logic) for Datalog (short DatalogMTL) [6] has been proposed as *a suitable extension to reason over the temporal domain.* DatalogMTL extends Datalog with rules such as $\boxminus_{[x,y]} A \rightarrow B$ or $\diamondminus_{[x,y]} A \rightarrow B$, with x, y being positive rational numbers, where B holds at time t if A holds at each (resp. some) time in the interval $[t-y, t-x]$.

To support the mentioned use cases, temporal reasoning must be enriched with the possibility to compute aggregations over time. Monotonic aggregation has received a great deal of attention in the literature [13, 16] and has been shown to be *a suitable extension of Datalog for aggregation.* For example, Shkapsky et al. [16] introduced rules of the form $A(g, x) \rightarrow B(g, msum(x))$, to express the monotonic sum of x values in A grouped by g. However, to the best of our knowledge, monotonic aggregation (or any other kind of temporal aggregation) in a temporal extension of Datalog has not been discussed so far.

Challenges. Monotonic aggregation over the temporal domain raises a number of technical challenges. Monotonic aggregation as defined for non-temporal reasoning is inefficient in memory and runtime performance over the temporal domain. In addition, forms of temporal-specific aggregation along the time axis are required, for example to retrieve the intervals where a value is monotonically increasing. Such operations should be non-blocking (i.e., return intermediate results without waiting for the entire aggregation to be complete) to sustain reactivity of the system.

Our contribution. In this paper, we investigate aggregation for temporal Datalog languages. Our main contributions are:

- We provide the first reasoning language that covers at the same time (1) full **recursion** (i.e., it comprises Datalog); (2) **aggregation** capabilities, and (3) **temporal** reasoning (i.e., it comprises MTL as in DatalogMTL).
- We provide a principled integration of state-of-the-art semantics of (1-3), overcoming critical semantic differences while sustaining efficient evaluation.
- We propose tractable, non-blocking algorithms for calculating monotonic *time-point aggregation* in Datalog. To this end, we augment specific fragments of DatalogMTL with monotonic aggregation, by using advanced interval tracking and enabling several time-based optimisations.
- We introduce specific, novel *monotonic aggregation over the time axis* without any pre-determined bounds, which requires special care as time is the aggregation dimension. In particular, we introduce, as an example, an operator to analyse *monotonic trends*.
- We provide *real-world examples* and show the practical relevance of our extensions within settings of industrial relevance, such as those in the financial settings experienced in our work with the Central Bank of Italy.

Organization. The remainder of this paper is organized as follows: In Section 2 we introduce some preliminaries for DatalogMTL and non-temporal monotonic aggregation. In Section 3 we discuss the requirements for temporal aggregation in Datalog. In Section 4 we present time-aware algorithms for time-point aggregation and discuss aggregation along the time axis in Section 5. We provide related work in Section 6 and conclude the paper in Section 7. We provide proofs and additional explanations in the appendix.

2 Preliminaries

In this section, we introduce DatalogMTL$^{\text{FP}}$ under continuous semantics [17, 18] as well as non-temporal aggregation as defined by Shkapsky et al. [16]. FP stands for *forward propagating* and refers to the fragment that does not allow reasoning backwards in time, as this is not required for most applications and has significant negative impact on complexity [19].

Syntax of DatalogMTL$^{\mathbf{FP}}$. Let $\mathbf{C}$ and $\mathbf{V}$ be disjoint sets of *constants*, and *variables*, respectively. A *term* is either a constant or a variable. An *atom* is an expression of the form $P(\boldsymbol{\tau})$, where P is a predicate of arity $n \geq 0$ and $\boldsymbol{\tau}$ is a n-tuple of terms. A *rule* is an expression of the form

$$A_1 \wedge \cdots \wedge A_k \rightarrow B \qquad\qquad \text{for } k \geq 0$$

where all A_i and B are literals that follow the grammar:

$$A \coloneqq P(\boldsymbol{\tau}) \mid \boxminus_\sigma A \mid \diamondminus_\sigma A \qquad\quad B \coloneqq P(\boldsymbol{\tau})$$

The conjunction of A_i is the rule *body*, whereas B is the rule *head*. Instead of $\wedge$ we sometimes use "," to denote conjunction. A predicate $p \in P$ is intensional (IDB), if p occurs in the head of the rule whose body is not empty, otherwise

p is extensional (EDB). A rule is *ground* if it contains no variables, and *safe* if each variable in the head is also contained in the body. A *program* Π is a finite set of safe rules and is in normal form if it contains only rules of the form:

$$P_1(\boldsymbol{\tau}_1) \wedge \cdots \wedge P_n(\boldsymbol{\tau}_n) \rightarrow P_0(\boldsymbol{\tau}_0) \qquad (n \geq 0) \tag{1}$$

$$\diamondminus_\sigma P_1(\boldsymbol{\tau}_1) \rightarrow P_0(\boldsymbol{\tau}_0) \tag{2}$$

$$\boxminus_\sigma P_1(\boldsymbol{\tau}_1) \rightarrow P_0(\boldsymbol{\tau}_0) \tag{3}$$

An *interval* $\sigma = \langle x, y \rangle$ is defined over the domain $\mathbb{Q}_{\geq 0} \cup \{+\infty\}$ where

$$\langle x, y \rangle = \{t \in \mathbb{Q}_{\geq 0} \mid x \leq t \leq y \text{ and } x, y \geq 0 \text{ and } x \neq t \text{ if } \langle \text{ is `(', and } y \neq t \text{ if } \rangle \text{ is `)'}\}.$$

We use $\langle$, (resp. $\rangle$) depending on the context for unspecified or value-specific, (, resp.) for open and [, resp.] for closed intervals. An interval σ is *bounded* if $x, y \in \mathbb{Q}_{\geq 0}$, *punctual* if it is of the form $[t, t]$, i.e., has a length of 0, and is denoted by t. The left endpoint is denoted by σ^- and the right endpoint by σ^+, where $\sigma^+ = \infty$ if the interval is unbounded to the right. The *length* is defined as $|\sigma| = \infty$ if $\sigma^+ = \infty$, otherwise $|\sigma| = \sigma^+ - \sigma^-$. For two non-empty intervals σ_1 and σ_2, we define the union $\sigma_1 \cup \sigma_2$ and intersection $\sigma_1 \cap \sigma_2$ as usual. Note that, a union operation creates one or two intervals, whereas an intersection of two intervals is either the empty set or exactly one interval. We define $\sigma_1 + \sigma_2$ as $\langle \sigma_1^- + \sigma_2^-, \sigma_1^+ + \sigma_2^+ \rangle$, where $\langle$ (resp. $\rangle$) is closed if σ_1 and σ_2 are left-closed (resp. right-closed), as well as $\sigma_1 \oplus \sigma_2$ as $\langle \sigma_1^- + \sigma_2^+, \sigma_1^+ + \sigma_2^- \rangle$, where $\langle$ (resp. $\rangle$) is closed if σ_1 is left-closed and σ_2 is right-open (resp. if σ_1 is right-closed and σ_2 is left-open). We define an interval as an empty set if $\sigma^- > \sigma^+$.

A *fact* is of the form $\alpha@t$, where α is a ground atom and t a punctual interval. A dataset D is a set of facts.

Semantics of DatalogMTL$^{\mathbf{FP}}$. Let $\mathfrak{M}$ be an interpretation based on a domain $\Delta \neq \emptyset$ for the variables and constants that specifies, for each ground atom α and each time point $t \in \mathbb{Q}_{\geq 0}$, whether α is true at t. In this case, we write $\mathfrak{M}, t \models \alpha$. Let ν be an assignment of elements of Δ to terms such that $\nu(c) = c$ for each constant c. We then define inductively:

$$\begin{array}{ll} \mathfrak{M}, t \models^\nu P(\boldsymbol{\tau}) & \text{if } \mathfrak{M}, t \models P(\nu(\boldsymbol{\tau})) \\ \mathfrak{M}, t \models^\nu \boxminus_\sigma A & \text{if } \mathfrak{M}, s \models^\nu A \text{ for all } s \text{ with } t - s \in \sigma \\ \mathfrak{M}, t \models^\nu \diamondminus_\sigma A & \text{if } \mathfrak{M}, s \models^\nu A \text{ for some } s \text{ with } t - s \in \sigma \end{array}$$

We say that $\mathfrak{M}$ satisfies a rule under an assignment ν if $\mathfrak{M}, t \models^\nu B$, whenever $\mathfrak{M}, t \models^\nu A_i$, for each $1 \leq i \leq k$, and satisfies a program if it satisfies all its rules. The interpretation $\mathfrak{M}$ is a model of Π and D if $\mathfrak{M}$ satisfies Π for all possible assignments (i.e., model of Π), and if $\mathfrak{M}, t \models^\nu \alpha$, for all facts $a@t \in D$ (i.e., model of D). Finally, a program Π and a dataset D entail a fact $\alpha@t$, written $(\Pi, D) \models \alpha@t$, if $\mathfrak{M}, t \models \alpha$ for each model $\mathfrak{M}$ of Π and D.

In the following, we provide as example the rules for detecting spamming traders to illustrate the usage of DatalogMTL. We define spamming traders as trader that send at least every 5 seconds a trading signal for a duration of one hour (assuming seconds as granularity), where X is the trader identifier.

$$\boxminus_{[0,3600)} \diamondminus_{[0,5)} \mathit{tradingSignal}(X) \rightarrow \mathit{spammingTrader}(X)$$

Monotonic Aggregation. Monotonic aggregation so far has only been introduced for non-temporal Datalog. Here we summarize the aggregation operators *mcount*, *msum*, *mmax*, *mmin*, as introduced by Shkapsky et al. [16]. The authors define head atoms of the form $P(k_1, \ldots, k_m, aggr\langle x, \mathbf{c_X}\rangle)$, where k_i are zero or more group-by terms and *aggr* is one of the four mentioned aggregation operations, where x is the aggregate term and $\mathbf{c_X}$, defined only for sum and count, is a sequence of contributor terms for the aggregation. The semantics of aggregates are operationally defined as the application of a mapping function from an input set or multiset G that represents the values for a single group-by key to an output set D. Given G, for *mmin* (resp. *mmax*) each element $g \in G$ is put into D, if g is smaller (greater) than the previously computed value. For *mcount* and *msum*, g is of the form (N_J, J), where N_J indicates the partial count/sum contributed by J, that is, N_J maps to x and J maps to $\mathbf{c_X}$. The aggregate functions now map the input set or multiset G to D by computing the sum over all maximum partial count N_J for all J and put the resulting value into D, in case the value is higher than the previously computed value. That is, the output set D consists of numbers (the (intermediary) results of the aggregation) that where inserted in a monotonically increasing order.

In the following, we provide as example the rules for path-counting in a DAG [16] to illustrate the usage of monotonic aggregation. The first rule counts each edge between two nodes as one path. The second rule counts the number of distinct paths between two nodes X and Y through every Z. The last rule derives the maximum value for each pair of nodes. For details of the execution of the rules, we refer to Shkapsky et al. [16].

$$\begin{aligned}
&\mathit{edge}(X,Y) \rightarrow \mathit{cpaths}(X,Y,\mathit{mcount}\langle(1,X)\rangle)\\
&\mathit{edge}(Z,Y), \mathit{cpaths}(X,Z,C) \rightarrow \mathit{cpaths}(X, Y, \mathit{mcount}\langle(C,Z)\rangle)\\
&\mathit{cpaths}(X,Y,C) \rightarrow \mathit{countpaths}(X, Y, \mathit{max}\langle(C)\rangle)
\end{aligned}$$

3 Requirements

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

- **RQ1: Declarative**. Managing temporal properties calls for complex interval checking logic and arithmetic. This easily leads to an error-prone and labor-intensive procedural approach. Declarative temporal operators that specify *what* should be done instead of *how* it should be done avoid such pitfalls.

Example. A baseline implementation for UC2 would first extend a time-point interval to an hour-long interval, 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.

- **RQ2: Implicit Time**. While explicit time in rules provides the user the possibility to access and modify temporal properties, they require semantic restrictions for the chosen operations to block arbitrary rule behavior. Implicit time handling, such as in temporal logics or DatalogMTL, does not have such issues, allowing for a fully declarative, composable solution.

Example. 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 ($P(S, E) \rightarrow R(S + 5, E + 5)$ *or* $P(S, E) \rightarrow R(S * S, E + 5)$)*.*

- **RQ3: Optimizability**. Temporal data often has the property of staying the same for a longer interval. Yet, many temporal operations are defined by time point. Declarative operators and implicit time already lead to a certain degree of optimizability, but any operators defined should take into account optimizable evaluation on intervals.

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

- **RQ4: Windowed 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.

- **RQ5: Fixed-interval 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.

- **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.

Looking at our archetypal use cases, UC2 is possible in a system that meets RQ4, UC1 in RQ5, and UC4 in RQ6. UC3 is already possible using recursion provided by Datalog, as well as, e.g., a system meeting RQ4 (it, however, would be hard to meet in a system that does not support recursion).

4 Time Point Aggregation

In this section, we take on the requirements for windowed and fixed-interval 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 windowed and fixed-interval aggregation and show efficient algorithms extending the application of the standard non-temporal aggregations [16] to the temporal context. Time axis aggregation (RQ6) is dealt with in the next section.

4.1 Moving Windows

The first type of aggregation we discuss are moving window aggregations (e.g., covering the last hour; RQ4). This form of aggregation aggregates 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$^{\text{FP}}$. Our basic approach is using the $\diamondminus$ operator to extend the time validity of facts to size w and then applying the non-temporal aggregation operation [16] for each time point.

Syntax. Let us start by defining the syntax of DatalogMTL$^{\text{FP}}$ with time point aggregation. For this, we extend the DatalogMTL$^{\text{FP}}$ normal form (1-3; cf. preliminaries) with additional time point aggregation rules of the following form:

$$x = aggr(P_1(\boldsymbol{\tau}_0, \boldsymbol{\tau}_1, a)) \rightarrow P_0(\boldsymbol{\tau}_0, x) \tag{4}$$

where *aggr* is the aggregation type (e.g., count, sum), P_1 and P_0 are predicates, $\boldsymbol{\tau}_0$ are the group-by attributes, $\boldsymbol{\tau}_1$ are the contributor terms for sum and count, and a the aggregate term (for count, where a is not part of the predicate, it should be assumed as 1 in the following algorithm). Note that P_1 has arity of size $|\boldsymbol{\tau}_0| + |\boldsymbol{\tau}_1|$, in case *aggr* is of type *count*, else $|\boldsymbol{\tau}_0| + |\boldsymbol{\tau}_1| + 1$ and P_2 has arity $|\boldsymbol{\tau}_0| + 1$.

Semantics. Let us continue by defining the semantics of the newly introduced rules. For this, we define the semantics on top of regular aggregation in Datalog [16] and apply it per time point:

$$\begin{aligned}
&\mathfrak{M}, t \models^{\nu} x = aggr(P(\boldsymbol{\tau_0}, \boldsymbol{\tau_1}, a)) \quad \text{ if} \\
&\quad \Pi = \{P(\boldsymbol{\tau_0}, \boldsymbol{\tau_1}, a) \rightarrow AggrResult(\boldsymbol{\tau_0}, aggr\langle a, \boldsymbol{\tau_1}\rangle)\} \text{ and} \\
&\quad S = \{P(\boldsymbol{\tau}) \mid \mathfrak{M}, t \models^{\nu} P(\boldsymbol{\tau})\} \text{ and } R = Eval(S, \Pi) \text{ and} \\
&\quad \nu(x) \in \{u \mid AggrResult(\boldsymbol{\tau_0}, u) \in R\}
\end{aligned}$$

where $\boldsymbol{\tau}$ stands for the sequence of terms $(\boldsymbol{\tau_0}, \boldsymbol{\tau_1}, a)$, *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 regular aggregation in Datalog.

Example. 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.

$$\diamondminus_{[0,3600)} Trade(u, id) \rightarrow TradeInterval(u, id) \tag{1}$$

$$m = mcount(TradeInterval(u, id)) \rightarrow NumberOfTrades(u, m) \tag{2}$$

Algorithm 1: Calculation of time point aggregation

```
Input: an aggregation type $T$, number of group-by terms $g$, and the
       aggregation predicate $A$, a set of $Facts$
Output: A set of aggregated facts $B$
1  $aggrResult := \emptyset$;
2  $cStorage := \emptyset$;
3  foreach $\alpha@\sigma$ in Facts; matching predicate of $A$ do
4      $a$, $groupByKey$, $cKey$ = getData($\alpha$, $A$, $T$, $g$);
5      $aggrVals = aggrResult[groupByKey]$;
6      $cVals = cStorage[groupByKey][cKey]$;
7      if $T = mmin$ then
8          UpdateList($aggrVals$, $a$, $\sigma^-$, $\sigma^+$, $(a,b) \Rightarrow min(a,b)$);
9      else if $T = mmax$ then
10         UpdateList($aggrVals$, $a$, $\sigma^-$, $\sigma^+$, $(a,b) \Rightarrow max(a,b)$);
11     else
12         $changes$ := UpdateList($cVals$, $a$,$\sigma^-$,$\sigma^+$,$(a,b) \Rightarrow max(a,b)$);
13         foreach $\{e, \sigma_2^-, \sigma_2^+\}$ in changes do
14             UpdateList($aggrVals$, $e$, $\sigma_2^-$, $\sigma_2^+$, $(a,b) \Rightarrow a + b$);
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. Note that we have provided a formulation using non-temporal Datalog in the Appendix, underpinning the highly increased readability of this version.

Algorithm. The application of the rules follows the standard chase procedure used with Datalog [12]. More specifically, we build upon the temporal work of Wałega et al. [19] who apply derivation rules for each rule in normal form plus additional merging rules for adjacent intervals exhaustively. Details on this algorithm are provided in extended form in the Appendix.

Our algorithm for time point aggregation extends existing temporal Datalog derivation rules as follows. Let $x = aggr(A(\tau_0, \tau_1, a)) \rightarrow B(\tau_0, x)$ be an instance of a rule of form 4. We apply Algorithm 1 to derive a set B of facts in the form $\beta@\sigma$. 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 B 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 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

Algorithm 2: Update List 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
```

groupByKey, the last term to *a* and the 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. 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). Since such value may increase over time, in order to avoid full recomputation, we just consider the difference with respect to the previous contributor value for a certain interval (Lines 13-14). At the end (Lines 14-16) we iterate over the lists and call mergeAdjacentInterval, which as the name says merge 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).

The UpdateList function is detailed in Algorithm 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). It then 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). Having reached the end of the list or an interval that is after the insertion range (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 harmonise the interval endpoints.

Theorem 1. *Algorithm 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 Sketch. We now show the basic idea behind the complexity of the algorithm. The interested reader can find the full proof in the Appendix. The $O(n)$ space requirement depends only on UpdateList. In particular, we argue that at most $2n$ intervals are created in the process of adding n intervals to the list.

This follows from a rather technical argument, but can be intuitively grasped as follows: Let σ denote an inserted interval. Our interval σ can intersect at its left endpoint (σ^-) one interval λ from the current list (and, importantly, at most one such interval, as maintaining non-overlapping intervals is central to the algorithm). In this case we need to split λ, creating a total of two new intervals: we replace λ by the interval λ^- to σ^-, create a new interval from σ^- to λ^+, and another new interval one from λ^+ to σ^+. Multiple other variants are possible. The interesting part is when our newly inserted interval σ intersects more than one interval, in which, intuitively speaking it uses "budget" stemming from earlier inserts that necessarily have not used their full "budget" of two inserts. As mentioned before, the full, technical argument can be found in the Appendix.

The runtime of $O(n^2)$ descends from the fact we iterate over all facts, and for each fact the UpdateList is performed, taking $O(n)$ time. Since both *changes* and *aggrVals* are sorted lists, they can be scanned together within a one-pass iteration, hence keeping linear complexity (intuitively, like in the usual merging of sorted lists).

We now show soundness and completeness of Algorithm 1, i.e., that every interval produced by the semantics is also derived by the algorithm, and the other way around, every interval derived by the algorithm is also produced by the semantics. Note that this is in the context of the facts that are input of the algorithm. As discussed earlier we consider recursive derivation in the Appendix.

Theorem 2. *Algorithm 1 is sound and complete.*

Proof Sketch. We now show the basic idea behind the soundness and completeness of the algorithm. The interested reader can find the full proof in the Appendix. Soundness requires to retrieve the best (e.g., min, max, highest) value for derived intervals. This is trivially achieved by Line 24 of UpdateList. Completeness requires that all intervals are derived. Since possible overlapping intervals at the start and end are split (Line 18-23), missing intervals before an entry are inserted (Line 14-17) and missing intervals after the last entry are added (Line 28-30), all possible intervals are created.

4.2 Fixed Intervals

The next type of aggregation we discuss are fixed interval aggregations (e.g., per month). In comparison to moving windows, we cannot rely on temporal operators provided by DatalogMTL due to different lengths, e.g., months vary between 28 and 31 days. Therefore, we introduce an additional 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. [5]).

Syntax. Let us start by formally defining the new operator. For this, we extend the DatalogMTL$^{\mathrm{FP}}$ normal form with an additional rule of the following form:

$$\vartriangle_{unit} P_1(\boldsymbol{\tau}_1) \rightarrow P_0(\boldsymbol{\tau}_0) \tag{5}$$

where $\vartriangle$ is the new time extension operator and *unit* is a time unit, e.g., day, month, or year.

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

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

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. Let us demonstrate the fixed interval aggregation by expressing UC1. Rule 1 extends the interval of the sales to its corresponding year and Rule 2 applies the sum operation, using the id (and price) as contributing terms, to derive the revenue of the year.

$$\vartriangle_{year} Sale(id, price) \rightarrow YearSale(id, price) \tag{1}$$
$$m = msum(\,YearSale(id, price)) \rightarrow Revenue(m) \tag{2}$$

Derivation Rule. In order to handle such rules in the chase procedure, we use the following derivation rule: If $\vartriangle_u \alpha \rightarrow \beta$ is a ground instance of a rule, and

$\alpha@\sigma_1 \in F$, then add $\beta@\sigma$ to F where $\sigma = [a, b)$, where $a = conv(conv(\sigma_1^-, u), u2)$, $b = conv(conv(\sigma_1^+, u) + 1\ u), u2)$ and $u2$ is the initial unit of σ_1, i.e., the time unit gets converted back to the precision of the predicate, e.g. 2021-02-16 with a precision of month gets after the inner conversion 2021-02 and after applying the outer conversion 2021-02-01.

Limitations. We briefly want to discuss the limitations of this approach. The main goal of using the extension of intervals is keeping compatibility with DatalogMTL and its interval semantics by calculating the aggregates over the interval boundaries asked in the query. However, using such an approach limits the possibility to answer queries that require changing time granularity, like in the following example where we wish to compare the revenue on a weekday basis. For such requirements, we suggest the introduction of unwrapping operations that map the timestamp to an appropriate representation in usual Datalog, e.g., by introducing rules of the form $Sale(id, price)@t \rightarrow SaleEscaped(id, price, weekday(t))$. A detailed discussion of such rules is out of scope for this paper.

5 Time Axis Aggregation

So far, we have focused on aggregations which work 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 summarize values changing over time considering adjacent intervals. One such function, which we study in detail in this paper, is the one detecting 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 formulation.

Example. Assume that we want to detect changes of control described in UC4. Then, we are interested in finding the time 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 $Shares(p, c, s)$ represents the number of shares s that an investor p owns of a company c. Rule 1 shows the *minc* operator in action. It takes 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.

$$\langle l, u \rangle = minc(Shares(p, c, s)) \rightarrow SInc(p, c, l, u) \tag{1}$$

Similar to the previous section, we (i) provide a normal form (syntax) for time axis aggregations, (ii) specify the semantics of each time axis aggregation, (iii) provide a derivation rule, and (iv) suggest an algorithm for time axis operation.

Syntax. We start by providing a normal form (actually already used in Rule (1), generalizing the normal form introduced for time point aggregation:

$$\langle \mathbf{x} \rangle = aggr(P_1(\boldsymbol{\tau}_0, \boldsymbol{\tau}_1, a)) \rightarrow P_0(\tau_0, \mathbf{x}) \tag{6}$$

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, $\boldsymbol{\tau_0}$ defines the group-by clause, $\boldsymbol{\tau_1}$ the contribution terms, not directly used in this form of aggregation 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 $\mathbf{x} = l, u$ to denote the lower and upper bound of the monotonic interval.

Semantics. We assume that the domain of temporal aggregation is that of disjoint intervals, and so, for a single group-by key, no ambiguity arises w.r.t. the aggregation term. Also, 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. Another effect of such disjoint intervals is that the semantics of time axis aggregation can be easily formulated by moving from time points to time intervals, i.e., $\mathfrak{M}, \sigma \models^{\nu} \phi$ if for all $t \in \sigma$ it holds that $\mathfrak{M}, t \models^{\nu} \phi$.

$$\begin{array}{ll}
\mathfrak{M}, \sigma \models^{\nu} \langle l, u \rangle = minc(P_1(\boldsymbol{\tau}_0, a)) & \text{if } \mathfrak{M}, \sigma \models^{\nu} M(P_1, \tau_0, l, u) \\
\mathfrak{M}, \sigma \models^{\nu} M(P_1, \tau_0, a, a) & \text{if } \mathfrak{M}, \sigma \models^{\nu} P_1(\tau_0, a) \\
\mathfrak{M}, \sigma \models^{\nu} M(P_1, \tau_0, l, u) & \text{if } \mathfrak{M}, \sigma_1 \models^{\nu} M(P_1, \tau_0, l, u_1) \text{ and} \\
 & \quad \mathfrak{M}, \sigma_2 \models^{\nu} M(P_1, \tau_0, l_2, u) \text{ and} \\
 & \quad u_1 \leq l_2 \text{ and } \sigma_1^+ \prec \sigma_2^- \text{ and } \sigma = \sigma_1 \cup \sigma_2
\end{array}$$

where $\prec$ is the predecessor relation (i.e., the intervals are adjacent), M a fresh predicate for deriving *minc*. In short, the second definition states that a value constant 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 $u_1 \geq l_2$.

Derivation rule. Let us now introduce our derivation mechanism. For each instance $\langle \mathbf{x} \rangle = aggr(A(\tau_0, \tau_1, a)) \rightarrow B(\tau_0, \mathbf{x})$ of a rule of form 6, where *aggr* is *minc* or *mdec* and τ_1 is the empty set, apply Algorithm 3 to derive a set B of facts of the form $\beta@\sigma$, where β contains the group-by values and the aggregation values l and u.

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 takes as input the aggregation predicate A and a number of group-by indices g; it 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

Algorithm 3: Calculation of monotonic monotonically increasing intervals

```
Input: number of group-by terms $g$ in $\alpha$, and the aggregation predicate $A$, a
       set of $Facts$
Output: A set of aggregated facts $B$
1  $B := \emptyset$;
2  foreach $\alpha@\sigma$ in Facts; matching predicate of $A$ do
3      $a$, $groupByKey := \text{getData}(\alpha,A,g)$;
4      $aggrGroup := X[groupByKey]$;
5      $nNode := aggrGroup.\text{insert}((a,a)\#\sigma)$;
6      $lNode := nNode.\text{previous}$;
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     $rNode := nNode.\text{next}$;
11     if $\sigma^+ = rNode.\sigma^- \wedge a \leq rNode.min$ then
12         Apply merging for $rNode$ similar to $lNode$;
13 return $X.values()$
```

$\leq$ to $\geq$. Line 13 returns all output facts of the algorithm (that is, it removes the required grouping for the calculation). Note that, similarly to Algorithm 1, delta-updates can be applied by skipping Line 1 of the algorithm.

Theorem 3. *Algorithm 3 has runtime $O(nlog(n))$ 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 Sketch. We show here the basic idea behind the complexity of the algorithm (details in Appendix). The space requirement of $O(n)$ depends on the fact that at most n entries are inserted in the tree. The runtime of $O(nlog(n))$ is based that we iterate over all facts ($O(n)$), and for each fact we insert to and remove entries from the tree, which has a complexity of $O(log(n))$.

Theorem 4. *Algorithm 3 is sound and complete.*

Proof Sketch. We show the basic idea behind the soundness and completeness regarding the maximum derived intervals (details in Appendix). Completeness follows immediately, since all intervals are inserted in the tree. For soundness we have to show that maximum intervals are derived with the correct lower and upper value. This follows by the two merging operations of possible adjacent intervals, where adjacent intervals are replaced with a new interval and the lowest value is chosen from the left and the highest value is chosen from the right interval.

6 Related Work

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$_{1S}$ [7, 14, 23]), the other including operators from temporal logic such as the always and eventually operator from LTL or CTL into Datalog (Templog [1], DatalogLite [9]). Newest developments are based on MTL [6, 17, 11], an extension of LTL to enrich the expressive power of Datalog programs. While most of the work for DatalogMTL so far purely consisted in complexity results, we find a proposal for a non-recursive fragment [6], as well as algorithm for stream reasoning by Wałega et al. [19]. The latter work clearly differs from the one we have presented in this paper, in that our focus is on monotonic aggregation over the temporal domain.

Similarly, aggregation in Datalog has been discussed for many years. The standard Datalog fixpoint semantics only works as long as rules define monotonic transformations (w.r.t. set containment). Aggregation breaks this requirement. Earlier solutions [15, 24] 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 [13]. Datalog^{FS} [13] 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.[16] provided first practical algorithms for monotonic aggregation by introducing contributor and group-by terms, which we used as foundations for our approach. While their work focused on optimizing aggregation for 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 [10], but only considers restricted use of sum and count by means of a sorted list of facts. Finally, Wang et al.[20] studied techniques to convert non-monotonic aggregates to monotonic ones.

Apart from Datalog-specific related work, LARS [2] is a temporal stream reasoning framework focusing on finite streams by extending propositional logic and can be seen as an extension for ASP. 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 perspecitve (instead of a query-centric), and only considers finite data. With a recent extension called weighted LARS [8], 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. In detail, the diamond-operator interprets the formula (i.e., all values of the sub-stream) by using the sum-operator of the semi-ring and the box-operator by using the product-operator of the semi-ring, while true values are mapped to one, false values are mapped to zero, *and* is mapped to times and *or* is mapped to plus. In comparison, we focus on combining two fundamental directions of

Datalog studies, namely DatalogMTL and monotonic aggregation and discuss efficient aggregation algorithms for DatalogMTL.

7 Conclusion

In this paper, we presented a solution for using monotonic aggregation over temporal Datalog fragments. This paper is, to the best of our knowledge, the first work to include (1) full recursion as in Datalog, (2) aggregation, and (3) temporal reasoning - a combination required by important use cases as we have shown. We showed that native temporal operators allow to apply specific optimization techniques to reduce performance overhead and memory consumption as well as increase usability. Our suggested algorithms can be executed in a non-blocking fashion (i.e., there is no need to wait for all the aggregation contributes to be available). In future work, we aim at providing a choice of experimental results showing the benefits of using native temporal operators for monotonic aggregation.

Acknowledgements

The financial support by the Vienna Science and Technology Fund (WWTF) grant VRG18-013 is gratefully acknowledged.

References

1. Abadi, M., Manna, Z.: Temporal logic programming. J. Symb. Comput. **8**(3), 277–295 (1989)
2. Beck, H., Dao-Tran, M., Eiter, T., Fink, M.: LARS: A logic-based framework for analyzing reasoning over streams. In: AAAI. pp. 1431–1438. AAAI Press (2015)
3. Bellomarini, L., Magnanimi, D., Nissl, M., Sallinger, E.: Neither in the programs nor in the data: Mining the hidden financial knowledge with knowledge graphs and reasoning. In: MIDAS@PKDD/ECML. Lecture Notes in Computer Science, vol. 12591, pp. 119–134. Springer (2020)
4. Benson, L., Grulich, P.M., Zeuch, S., Markl, V., Rabl, T.: Disco: Efficient distributed window aggregation. In: EDBT. pp. 423–426. OpenProceedings.org (2020)
5. Bettini, C., Jajodia, S., Wang, X.S.: Time granularities in databases, data mining, and temporal reasoning. Springer (2000)
6. Brandt, S., Kalayci, E.G., Kontchakov, R., Ryzhikov, V., Xiao, G., Zakharyaschev, M.: Ontology-based data access with a Horn fragment of metric temporal logic. In: AAAI. pp. 1070–1076. AAAI Press (2017)
7. Chomicki, J.: Polynomial time query processing in temporal deductive databases. In: PODS. pp. 379–391. ACM Press (1990)
8. Eiter, T., Kiesel, R.: Weighted LARS for quantitative stream reasoning. In: ECAI. Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 729–736. IOS Press (2020)
9. Gottlob, G., Grädel, E., Veith, H.: Datalog LITE: a deductive query language with linear time model checking. ACM Trans. Comput. Log. **3**(1), 42–79 (2002)

10. Kaminski, M., Grau, B.C., Kostylev, E.V., Horrocks, I.: Complexity and expressive power of disjunction and negation in limit datalog. In: AAAI. pp. 2862–2869 (2020)
11. Kikot, S., Ryzhikov, V., Walega, P.A., Zakharyaschev, M.: On the data complexity of ontology-mediated queries with MTL operators over timed words. In: Description Logics. CEUR Workshop Proceedings, vol. 2211. CEUR-WS.org (2018)
12. Maier, D., Mendelzon, A.O., Sagiv, Y.: Testing implications of data dependencies. ACM Trans. Database Syst. **4**(4), 455–469 (1979)
13. Mazuran, M., Serra, E., Zaniolo, C.: Extending the power of datalog recursion. VLDB J. **22**(4), 471–493 (2013)
14. Ronca, A., Kaminski, M., Grau, B.C., Motik, B., Horrocks, I.: Stream reasoning in temporal datalog. In: AAAI. pp. 1941–1948. AAAI Press (2018)
15. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive databases. In: PODS. pp. 114–126. ACM Press (1992)
16. Shkapsky, A., Yang, M., Zaniolo, C.: Optimizing recursive queries with monotonic aggregates in deals. In: ICDE. pp. 867–878. IEEE Computer Society (2015)
17. Walega, P.A., Grau, B.C., Kaminski, M., Kostylev, E.V.: Datalogmtl: Computational complexity and expressive power. In: IJCAI. pp. 1886–1892. ijcai.org (2019)
18. Walega, P.A., Grau, B.C., Kaminski, M., Kostylev, E.V.: Tractable fragments of datalog with metric temporal operators. In: IJCAI. pp. 1919–1925. ijcai.org (2020)
19. Walega, P.A., Kaminski, M., Grau, B.C.: Reasoning over streaming data in metric temporal datalog. In: AAAI. pp. 3092–3099. AAAI Press (2019)
20. Wang, Q., Zhang, Y., Wang, H., Geng, L., Lee, R., Zhang, X., Yu, G.: Automating incremental and asynchronous evaluation for recursive aggregate data processing. In: SIGMOD Conference. pp. 2439–2454. ACM (2020)
21. Xu, J., Liu, C., Zhao, X., Yongchareon, S., Ding, Z.: Resource management for business process scheduling in the presence of availability constraints. ACM Trans. Manag. Inf. Syst. **7**(3), 9:1–9:26 (2016)
22. Yen, T., Oprea, A., Onarlioglu, K., Leetham, T., Robertson, W.K., Juels, A., Kirda, E.: Beehive: large-scale log analysis for detecting suspicious activity in enterprise networks. In: ACSAC. pp. 199–208. ACM (2013)
23. Zaniolo, C.: Logical foundations of continuous query languages for data streams. In: Datalog. Lecture Notes in Computer Science, vol. 7494. Springer (2012)
24. Zaniolo, C., Arni, N., Ong, K.: Negation and aggregates in recursive rules: the LDL++ approach. In: DOOD. LNCS, vol. 760, pp. 204–221. Springer (1993)

A Overview of Appendix

The appendix is structured as follows: In Section B we provide an extended DatalogMTL reasoning algorithm, in Section C we provide additional background information to Section 4, and in Section D we explain details for Section 5.

B DatalogMTL Reasoning Algorithm

In this section we present a modified version of the DatalogMTL reasoning algorithm presented in [19]. This version modifies and extends the derivation rules with the newly introduced rules and establishes new output possibilities. Critically, existing approaches are specific to streaming data and only allow to output derived facts for pre-chosen time points, requiring to decide before knowing the reasoning result which time points contain relevant information.

Algorithm 4: A reasoning algorithm for monotonic aggregation over time by using DatalogMTL$^{\mathrm{FP}}$

```
Input: A program $\Pi$, a dataset $S$, a list of output predicates $Q$, a set of
       output intervals $X$ (optional, if not present $X = \{D\}$), a bounded
       reasoning interval domain $D$
Output: All valid predicates, given in $Q$, with its intervals in $D$ and $X$
1 $F := S$;
2 do
3     Apply rules from Table 1 exhaustively;
4     $F := F \cap D$, i.e., map each $P(\boldsymbol{\tau})@\sigma_f \in F$ to $P(\boldsymbol{\tau})@\sigma$, s.t., $\sigma = \sigma_f \cap D$;
5 while $F$ has changed;
6 foreach $P(\boldsymbol{\tau})$ in $F$ do
7     if $P$ in $Q$ then
8         $P'(\boldsymbol{\tau}) \leftarrow P(\boldsymbol{\tau}) \cap X$, i.e., map each $\sigma_x \in X$ and $P(\boldsymbol{\tau})@\sigma_f$ to $P'(\boldsymbol{\tau})@\sigma$
            s.t. $\sigma = \sigma_x \cap \sigma_f$ is not empty;
9         output $P'(\boldsymbol{\tau})$;
```

Algorithm 4 takes as input a program Π in normal form, a dataset S, a list of output predicates Q, optionally a set of output intervals X, a bounded reasoning interval domain D and returns as output a maximal set of facts (i.e., due to the union rule all intervals are merged, if possible). Only facts valid in the reasoning domain interval D are returned, and, if X is given, in particular, only those valid within non-empty intersection with some interval $\sigma_x \in X$. Defining X as a single punctual interval allows to output all valid facts at a certain time point. Line 1 initializes the derived facts with the provided dataset. Lines 2-5 apply the rules of Table 1 to derive new facts and restrict those facts to the chosen interval domain.

- The **H**orn rule adds all heads, where the intervals of the bounded body predicates overlap.

(H) If $\wedge_{i=1}^{n}\alpha_i \rightarrow \beta$ a ground instance of a rule and for each i $\alpha_i@\sigma_i \in F$, then add $\beta@\sigma$ to F with $\sigma = \cap_{i=1}^{n}\sigma_i$

(D) If $\diamondminus_{\sigma_1}\alpha \rightarrow \beta$ a ground instance of a rule, and $\alpha_2@\sigma_2 \in F$, then add $\beta@\sigma$ to F where $\sigma = \sigma_1 + \sigma_2$.

(B) If $\boxminus_{\sigma_1}\alpha \rightarrow \beta$ a ground instance of a rule and $\alpha_2@\sigma_2 \in F$, then add $\beta@\sigma$ to F where $\sigma = \sigma_1 \oplus \sigma_2$.

(T) If $\triangle_{unit}\alpha \rightarrow \beta$ a ground instance of a rule, and $\alpha@\sigma_1 \in F$, then add $\beta@\sigma$ to F where $\sigma = [a, b)$, where $a = conv(conv(\sigma_1^-, u), u2)$, $b = conv(conv(\sigma_1^+, u) + 1\ u), u2)$, and $u2$ is the original unit of σ_1.

(P) If $x = aggr(A(\tau_0, \tau_1, a)) \rightarrow B(\tau_0, x)$ an instance of a rule, then apply Algorithm 1 to derive a set X of facts in the form $\beta@\sigma$, which are added to F.

(A) If $\langle \mathbf{x} \rangle = aggrT(A(\tau_0, a)) \rightarrow B(\tau_0, \mathbf{x})$, an instance of a rule, then apply Algorithm 3 to derive a set X of facts of the form $\beta@\sigma$, which are added to F.

(U) If $\alpha@\sigma_1, \alpha@\sigma_2$ in F, and $\sigma_1 \cup \sigma_2$ coincides with one single interval σ, then remove $\alpha@\sigma_1, \alpha@\sigma_2$ from F and add $\alpha@\sigma$ to F.

Table 1. Derivation Rules

- The **D**iamond rule captures the semantics of the $\diamondminus$ operator. It expands (i.e., if $\sigma_1^- < \sigma_1^+$) the interval σ_2 and shifts (i.e., if $\sigma_1^- > 0$) it into the future.
- The **B**ox rule captures the semantic of the $\boxminus$ operator. It reduces (i.e., if $\sigma_1^- < \sigma_1^+$) the interval σ_2 and shifts (i.e., if $\sigma_1^- > 0$) it into the future.
- The **T**riangle rule captures the semantics of the $\triangle$ operator. It expands the interval to the range of the chosen unit.
- The time **P**oint aggregation rule captures the semantics of time point aggregation. It applies the aggregation per time point over all facts where the time point is in the interval of the fact.
- The time **A**xis aggregation rule captures the semantics of monotonic increasing and decreasing intervals. It applies the aggregation over the time axis and merges adjacent intervals following the given criteria.
- The **U**nion rule derives new intervals based on derived facts by merging overlapping intervals. In addition, it optimizes the storage size by removing the old, now merged, intervals. This rule includes the deletion of subsets.

Lines 6-9 output the derived facts that are part of the list of predicates Q and intersect with the output intervals X. Due to the union rule, each interval of a fact in the output is not a subset of another interval of the same fact in the output. Note that this algorithm does not terminate, in case the time point aggregation derives no fixpoint, that is for example when a recursive sum is calculated, where additional facts are produced per application of the rule. This is a typical issue for monotonic aggregation, which is also experienced in [16].

C Appendix for Section 3

In this section we provide a non-temporal Datalog encoding of UC2 and the proofs of Theorem 1 and Theorem 2.

$$\mathit{Trade}(u, id, t) \rightarrow \mathit{TradeInterval}(u, id, t, t + 3600, 1, 0) \tag{1}$$

$$\mathit{TradeInterval}(u, id, x, y, x_b, y_b) \rightarrow \mathit{IPoint}(x, x_b), \mathit{IPoint}(y, y_b) \tag{2}$$

$$\mathit{IPoint}(t, c) \rightarrow \mathit{Int}'(0, \mathit{mmin}(t), 1, c) \tag{3a}$$

$$\mathit{Int}'(x, y, 1, 0) \rightarrow \mathit{Int}(x, y, 1, 1) \tag{3b}$$

$$\mathit{Int}'(x, y, 1, 1) \rightarrow \mathit{Int}(x, y, 1, 0) \tag{3c}$$

$$\mathit{Int}(\_, t_p, \_, 1), \mathit{IPoint}(t, c), t > t_p \rightarrow \mathit{Int}'(t_p, \mathit{mmin}(t), 0, c) \tag{4a}$$

$$\mathit{Int}(\_, t_p, \_, 0), \mathit{IPoint}(t, c), t > t_p \rightarrow \mathit{Int}'(t_p, \mathit{mmin}(t), 1, c) \tag{4b}$$

$$\mathit{Int}'(x, y, 0, 0) \rightarrow \mathit{Int}(x, y, 0, 1) \tag{4c}$$

$$\mathit{Int}'(x, y, 0, 1) \rightarrow \mathit{Int}(x, y, 0, 0) \tag{4d}$$

Note that Rules 3b and 3c complete 4c and 4d

$$\begin{aligned} &\mathit{Int}(x, y, xc, yc), \mathit{TradeInterval}(u, id, st, et, sc, ec), \\ &(x < et), (y > st), \rightarrow V(u, id, x, y, xc, yc) \end{aligned} \tag{5a}$$

$$\begin{aligned} &\mathit{Int}(x, y, xc, yc), \mathit{TradeInterval}(u, id, st, et, sc, ec), \\ &(x = et \wedge xc = 1 \wedge ec = 1), \\ &(y > st) \rightarrow V(u, id, x, y, xc, yc) \end{aligned} \tag{5b}$$

$$\begin{aligned} &\mathit{Int}(x, y, xc, yc), \mathit{TradeInterval}(u, id, st, et, sc, ec), \\ &(x < et), (y = st \wedge yc = 1 \wedge sc = 1) \\ &\rightarrow V(u, id, x, y, xc, yc) \end{aligned} \tag{5c}$$

$$\begin{aligned} &\mathit{Int}(x, y, xc, yc), \mathit{TradeInterval}(u, id, st, et, sc, ec), \\ &(x = et \wedge xc = 1 \wedge ec = 1), \\ &(y = st \wedge yc = 1 \wedge sc = 1), \\ &\rightarrow V(u, id, x, y, xc, yc) \end{aligned} \tag{5d}$$

$$V(u, id, x, y, xc, yc) \rightarrow \mathit{NumberOfTrades}(u, \mathit{mcount}(id), x, y, xc, yc) \tag{6}$$

Fig. 1. UC2 in non-temporal Datalog

UC2 in non-temporal Datalog. Figure 1 highlights the required rules for writing such an aggregation in non-temporal Datalog. In a first step, for all four aggregation types, we calculate all interval points at which the query answer may change. We provide the calculation of such points in Rules 1-4. Rule 1 extends the *Trade* facts to one-hours 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. Having established the relevant intervals, 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

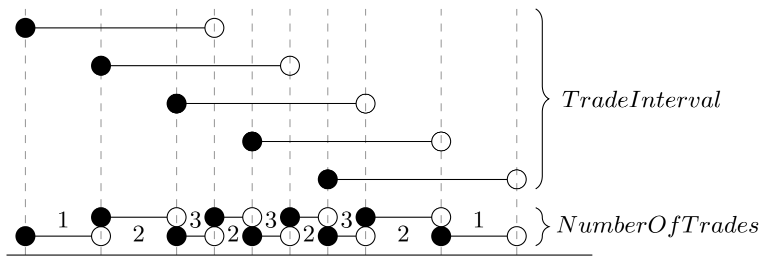

Fig. 2. Example for number of trades per hour

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.

Consider Figure 2 for an example of five *Trade* events already extended to one hour intervals. The five top-most intervals denote all *TradeInterval* predicates. Vertical, dashed lines show all *IPoint*. The intervals at the bottom of Figure 2 refer to the final result, that is, the *NumberOfTrades* predicates.

Theorem 1 *Algorithm 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 require $O(2n)$ times 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 a 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 run 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 2 *Algorithm 1 is sound and complete.*

Proof. Let us start with the soundness of the algorithm, which is trivial. We have to show that the returned results are the best result per interval, where best mean the minimal one for *mmin*, the maximum one for *mmax*, the highest sum for *msum* or count for *mcount*. This means, for each interval we have in the output list, we detected the best value. This is given by the chosen *aggr*-function and Line 24 of UpdateList that calculates the best value for each interval. Let us now consider the completeness of the algorithm to show that all interval rules are considered. For this, we show that we do not miss any interval in the update process, when we consider a new fact. It is clear by Line 14-17 that an interval that does not exist before a list entry, is added and by Line 28-30 that an interval that is after the last list entry or together with Line 11-13 ends after an existing list entry is added. It remains to show that also existing intervals are modified correctly. This is handled by Lines 18-20 if the interval starts inside another interval and Lines 21-23 if the interval ends inside another interval. The remaining part of time point algorithm does not modify the intervals, except of the merging functions, which only combine existing intervals to maximum ones, having no impact on the covering of the intervals. Hence, the algorithm is sound and complete.

D Appendix for Section 4

In this section, we show the proofs of Theorem 3 and Theorem 4.

Theorem 3 *Algorithm 3 has runtime $O(nlog(n))$ 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 4 *Algorithm 3 is sound and complete.*

Proof. Let us start by 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 neighbour. Then we are done and we have the maximum monotonic increasing interval.
- The interval is adjacent to the left neighbour. If the two monotonic intervals are increasing, then by Line 7-9, we merge the two neighbours creating a new bigger maximal interval over the currently maximal interval of the left neighbour and the added interval and update the bounds.
- The interval is adjacent to the right neighbour. We have two cases, either the inserted interval is not adjacent with the left neighbour, or the interval is also adjacent with the left neighbour. 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 neighbour 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, the derived interval is always the maximum interval and the algorithm is sound and complete.