Noname manuscript No.
(will be inserted by the editor)

Scalable Online First-Order Monitoring

Joshua Schneider · David Basin · Frederik Brix · Srđan Krstić · Dmitriy Traytel

the date of receipt and acceptance should be inserted later

Abstract Online monitoring is the task of identifying complex temporal patterns while incrementally processing streams of data-carrying events. Existing state-of-the-art monitors for first-order patterns, which may refer to and quantify over data values, can process streams of modest velocity in real-time. We show how to scale up first-order monitoring to substantially higher velocities by slicing the stream, based on the events' data values, into substreams that can be monitored independently. Because monitoring is not embarrassingly parallel in general, slicing can lead to data duplication. To reduce this overhead, we adapt hash-based partitioning techniques from databases to the monitoring setting. We implement these techniques in an automatic data slicer based on Apache Flink and empirically evaluate its performance using two tools—MonPoly and DejaVu—to monitor the substreams. Our evaluation attests to substantial scalability improvements for both tools.

Keywords Runtime Verification · Online Monitoring · Temporal Logic · Data Parallelism

1 Introduction

In large-scale software systems, millions of events occur each second [26, 42]. Identifying instances of interesting patterns in these high-velocity data streams is a central challenge in the area of runtime verification and monitoring. Often, this search must be performed *online* given the systems' continuous operation and the massive amounts of data they produce.

An *online monitor* takes as input a pattern and a stream of data, which it consumes incrementally, and it detects and outputs matches with the pattern. The specification language for patterns significantly influences the monitor's time and space complexity. For propositional languages, such as metric temporal logic or metric dynamic logic, existing state-of-the-art monitors are capable of handling millions of events per second in real time on commodity hardware [10, 17, 47, 48]. Propositional languages, however, are severely limited in their expressiveness. Since they regard events as atomic, they cannot formulate dependencies between the data values stored in events. First-order languages, such as metric first-order temporal logic (MFOTL) [15], do not have this limitation. Various online monitors [7, 9, 15, 18, 37, 49, 51] can handle first-order languages for event streams, but only with modest velocities.

We improve the scalability of online first-order monitors using parallelization. There are two basic approaches regarding what to parallelize. *Task parallelism* adapts the monitoring algorithm to evaluate multiple subpatterns in parallel. The amount of parallelization offered is limited by the number of subpatterns of a given input pattern. The alternative is *data parallelism*, where multiple copies of the monitoring algorithm are run unchanged as a black box, in parallel, on different portions of the input data stream.

In this article we focus on data parallelism, which is attractive for several reasons. As it is a black-box approach, data parallelism allows us to reuse existing monitors, which implement heavily optimized sequential algorithms. It also offers a virtually unbounded amount of parallelization, especially on high-volume and high-velocity data streams. Finally, it caters for the use of general-purpose libraries for data-parallel stream processing. These libraries deal with

Joshua Schneider is supported by the US Air Force grant "Monitoring at Any Cost" (FA9550-17-1-0306). Srđan Krstić is supported by the Swiss National Science Foundation grant "Big Data Monitoring" (167162).

J. Schneider · D. Basin · F. Brix · S. Krstić
Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
E-mail: {joshua.schneider, srdan.krstic}@inf.ethz.ch

D. Traytel
Department of Computer Science, University of Copenhagen, Denmark
E-mail: traytel@di.ku.dk

common challenges in high-performance computing, such as deployment on computing clusters, fault-tolerance, and back-pressure induced by velocity spikes.

Data parallelism has previously been used to scale up the offline monitoring of systems (Section 2), which is performed after the systems completed their execution. Yet neither offline nor online monitoring is an *embarrassingly parallel* task in general. Thus, in some cases, the monitors executing in parallel must synchronize. Alternatively, careful data duplication across these monitors allows for a non-blocking parallel architecture. An important contribution of prior work on scalable offline monitoring is the development of a (*data*) *slicing framework* [11]. The framework takes as input an MFOTL formula (Section 3) and a splitting strategy that determines to which of the parallel monitors the data should be sent. The framework's output is a dispatcher that forwards events to appropriate monitors and ensures that the overall parallel architecture collectively produces exactly the same results that a single monitor would produce.

The previous slicing framework has three severe limitations. First, data can be sliced on only one free variable at a time. Although it is possible to compose multiple single-variable slices into multi-variable slices, this composition is less expressive than simultaneously slicing on multiple variables. We explain the difference in Section 4.3. Second, the user of the slicing framework must supply a splitting strategy, even when it is obvious what the best strategy is for the given formula. Third, the framework's implementation uses Google's MapReduce library for parallel processing, which restricts its applicability to just offline monitoring.

This article addresses all of the above limitations and thereby makes the following contributions:

- We generalize the offline slicing framework [11] to support simultaneous slicing on multiple variables and we also adapt it to online monitoring (Section 4).
- We instantiate the slicing framework with an automatic splitting strategy (Section 5) inspired by hash-based partitioning and the hypercube algorithm [4, 39]. This algorithm has previously been used to parallelize relational join operators in databases. *Skew*, which is the presence of frequently occurring values, can cause imbalances in hash-based partitioning. Our automatic strategy also addresses this issue by separately handling events with frequently occurring values, using another database technique that we adapt to the monitoring setting.
- We implement our new slicing framework using the Apache Flink [5] stream processing engine (Section 6). We use both MonPoly [15, 16] and DejaVu [37] as black-box monitors for the slices. A particular challenge was to efficiently checkpoint MonPoly's state within Flink to achieve fault-tolerance. (We do not address fault-tolerance and skew for DejaVu.)
- We evaluate the slicing framework and automatic strategy selection on both real-world data based on Nokia's data collection campaign [14] and synthetic data exercising difficult cases (Section 7). We show that the overall parallel architecture substantially improves the throughput. Although the optimality of the hypercube approach in terms of a balanced data distribution is out of reach for general MFOTL formulas, we demonstrate that our automatic splitting results in balanced slices and improved monitoring performance.

An earlier version of this work was presented at RV 2018 [52]. This article extends the conference paper with detailed proofs of the slicing framework's correctness (Section 4) and a significantly expanded description of the automatic strategy selection algorithm (Section 5). This includes background information on the standard hypercube algorithm from databases (Section 5.1), which we build upon. Moreover, we have integrated DejaVu as a second black-box monitor in addition to MonPoly in our Apache Flink-based implementation (Section 6). This demonstrates our framework's generality. Finally, we (re-)evaluate both versions of the resulting parallel online monitor (Section 7). For both, higher parallelism yields significantly improved performance.

All theorems stated in this article, namely those establishing our slicing framework's correctness, have been mechanically checked using the Isabelle proof assistant. Additionally, we provide detailed proofs in this article for the benefit of readers not familiar with Isabelle. Both our implementation [54] and formalization [56] are publicly available. The formal verification of an MFOTL monitor modeled after MonPoly has been addressed in a separate line of work [12, 55].

2 Related Work

Our work builds on the slicing framework introduced by Basin et al. [11]. This framework ensures the sound and complete slicing of the event stream with respect to MFOTL formulas. It prescribes the use of composable operators, called slicers, that slice data associated with a single free variable, or slice data based on time. As explained in the introduction, we have generalized their data slicers to operate simultaneously on all free variables in a formula. Moreover, the use of MapReduce in the original framework's implementation limited it to offline monitoring. In contrast, our Apache Flink implementation supports online monitoring. Finally, our implementation extends the framework with an automatic strategy selection that results in a balanced load distribution for the slices in our empirical evaluation.

Barre et al. [6], Bianculli et al. [23], and Bersani et al. [22] use task parallelism over subformulas to parallelize propositional offline monitors. The degree of parallelization in these approaches is limited by the formula's size.

Parametric trace slicing [51] lifts propositional monitoring to parametric specifications. To this end, a trace with parametric events is split into propositional slices with events grouped by their parameter instances, which can be monitored independently. Parametric trace slicing considers only non-metric policies with top-level universal quantification. Barringer et al. [7] generalize this approach to more complex properties expressed using quantified event automata (QEA). Reger and Rydeheard [49] delimit the *sliceable* fragment of first-order linear temporal logic (FO-LTL) that admits a sound application of parametric trace slicing. The fragment prohibits deeply nested quantification and using the "next" operator. These restrictions originate from the time model used, in which time-points consist of exactly one event. Hence, when an event is removed from a slice, information about that time-point is lost. Our time model, based on sequences of time-stamped *sets* of events, avoids such pitfalls. Parametric trace slicing produces an exponential number of propositional slices (in the domain's size), whereas we use as many slices as there are parallel monitors available.

Kuhtz and Finkbeiner [40] show that the LTL monitoring problem belongs to the complexity class AC^1(logDCFL) and hence can be efficiently parallelized. However, the Boolean circuits used to establish the lower bound must be built for each trace in advance, which limits these results to offline monitoring. A similar limitation applies to the work by Bundala and Ouaknine [24] and Feng et al. [32], who study variants of MTL and TPTL.

Complex event processing (CEP) systems analyze streams by recognizing composite events as (temporal) patterns built from simple events. These systems allow for ample parallelism. However, their languages are often based on SQL extensions without a clear semantics. An exception is BeepBeep [34,35]: a multi-threaded stream processor that supports LTL-FO$^+$, a first-order variant of LTL. The parallelism in BeepBeep must, however, be arranged manually by the user.

Event stream processing systems have been extensively studied in the database community. We focus on the most closely related works. The hypercube algorithm (also known as the shares algorithm) was proposed by Afrati and Ullman [4] in the context of MapReduce. The algorithm is similar to the triangle counting algorithm by Suri and Vassilvitskii [57] and can be traced back to the parallel evaluation of datalog queries [33]. The hypercube algorithm is optimal for conjunctive queries with one communication round on skew-free databases [21], which do not contain heavy hitters (data values that occur more frequently than a fixed threshold).

The hypercube algorithm and other hash-based partitioning schemes are sensitive to skew. Rivetti et al. [50] suggest applying a greedy balancing strategy after identifying heavy hitters. This approach is restricted to conjunctive queries where all relations share a common join key. Joglekar et al. [38] improve asymptotically over the hypercube algorithm by using multiple communication rounds. Nasir et al. [43,44] balance skew for associative stream operators without explicitly identifying heavy hitters. Vitorovic et al. [58] combine the hash-based hypercube, prone to heavy hitters, with random partitioning [45], resilient to heavy hitters. Their combination only applies to conjunctive queries and limits the impact of skew without improving the worst-case performance. All these approaches are unsuitable for handling MFOTL formulas. Instead we follow a hypercube variant that is worst-case optimal in the presence of skew [39]. The heavy hitters must be known in advance in this approach. In contrast to the earlier algorithm by Beame et al. [20], it is sufficient to consider the heavy hitters of each attribute in isolation.

3 Metric First-Order Temporal Logic

We briefly recall the syntax and semantics of our specification language, metric first-order temporal logic (MFOTL) [15].

We fix a set of *names* $\mathbb{E}$ and for simplicity assume a single infinite *domain* $\mathbb{D}$ of values. The names $r \in \mathbb{E}$ have associated arities $\iota(r) \in \mathbb{N}$. An *event* $r(d_1,\dots,d_{\iota(r)})$ is an element of $\mathbb{E} \times \mathbb{D}^*$. We call $1,\dots,\iota(r)$ the *attributes* of the name r. We further fix an infinite set $\mathbb{V}$ of variables, such that $\mathbb{V}$, $\mathbb{D}$, and $\mathbb{E}$ are pairwise disjoint. Let $\mathbb{I}$ be the set of nonempty intervals $[a,b) := \{x \in \mathbb{N} \mid a \leq x < b\}$, where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$ and $a < b$. Formulas φ are constructed inductively, where t_i, r, x, and I range over $\mathbb{V} \cup \mathbb{D}$, $\mathbb{E}$, $\mathbb{V}$, and $\mathbb{I}$, respectively:

$$\varphi ::= r(t_1,\dots,t_{\iota(r)}) \mid t_1 \approx t_2 \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x. \varphi \mid \newline \bullet_I \varphi \mid \bigcirc_I \varphi \mid \varphi \,\mathsf{S}_I\, \varphi \mid \varphi \,\mathsf{U}_I\, \varphi.$$

Along with the Boolean operators, MFOTL includes the metric past and future temporal operators $\bullet$ (*previous*), S (*since*), $\bigcirc$ (*next*), and U (*until*), which may be nested freely. We define other standard operators in terms of this minimal syntax: truth $\top := \exists x.\ x \approx x$, falsehood $\bot := \neg\top$, inequality $t_1 \not\approx t_2 := \neg(t_1 \approx t_2)$, conjunction $\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$, universal quantification $\forall x.\ \varphi := \neg(\exists x.\ \neg\varphi)$, eventually $\Diamond_I \varphi := \top \,\mathsf{U}_I\, \varphi$, always $\Box_I \varphi := \neg \Diamond_I \neg\varphi$, once $\blacklozenge_I \varphi := \top \,\mathsf{S}_I\, \varphi$, and historically (always in the past) $\blacksquare_I \varphi := \neg \blacklozenge_I \neg\varphi$. We write $V(\varphi)$ for the set of free variables of the formula φ.

MFOTL formulas are interpreted over streams of time-stamped events. We group finite sets of events that happen concurrently (from the event source's point of view) into *databases*. An (*event*) *stream* ρ is an infinite sequence $(\tau_i, D_i)_{i\in\mathbb{N}}$ of databases D_i with associated time-stamps τ_i. We assume discrete time-stamps, modeled as natural numbers $\tau \in \mathbb{N}$. The event source may use a finer notion of time than the one used for time-stamps: databases at different indices $i \neq j$ may have the same time-stamp $\tau_i = \tau_j$. The sequence of time-stamps must be non-strictly increasing ($\forall i.\ \tau_i \leq \tau_{i+1}$) and always eventually strictly increasing ($\forall \tau.\ \exists i.\ \tau < \tau_i$).

The relation $v, i \models_\rho \varphi$ (Figure 1) defines the satisfaction of the formula φ for a valuation v at an index i with respect to the

$v, i \models_\rho r(t_1, \dots, t_{\iota(r)})$	if $r(v(t_1), \dots v(t_{\iota(r)})) \in D_i$
$v, i \models_\rho t_1 \approx t_2$	if $v(t_1) = v(t_2)$
$v, i \models_\rho \neg\varphi$	if $v, i \not\models_\rho \varphi$
$v, i \models_\rho \varphi \vee \psi$	if $v, i \models_\rho \varphi$ or $v, i \models_\rho \psi$
$v, i \models_\rho \exists x.\ \varphi$	if $v[x \mapsto z], i \models_\rho \varphi$ for some $z \in \mathbb{D}$
$v, i \models_\rho \bullet_I \varphi$	if $i > 0$, $\tau_i - \tau_{i-1} \in I$, and $v, i-1 \models_\rho \varphi$
$v, i \models_\rho \bigcirc_I \varphi$	if $\tau_{i+1} - \tau_i \in I$ and $v, i+1 \models_\rho \varphi$
$v, i \models_\rho \varphi \mathsf{S}_I \psi$	if $v, j \models_\rho \psi$ for some $j \leq i$, $\tau_i - \tau_j \in I$, and $v, k \models_\rho \varphi$ for all k with $j < k \leq i$
$v, i \models_\rho \varphi \mathsf{U}_I \psi$	if $v, j \models_\rho \psi$ for some $j \geq i$, $\tau_j - \tau_i \in I$, and $v, k \models_\rho \varphi$ for all k with $i \leq k < j$

Fig. 1 Semantics of MFOTL

stream $\rho = (\tau_i, D_i)_{i \in \mathbb{N}}$. The valuation v is a mapping $V(\varphi) \to \mathbb{D}$, assigning domain elements to the free variables of φ. Overloading notation, v is also the extension of v to the domain $V(\varphi) \cup \mathbb{D}$, setting $v(t) = t$ whenever $t \in \mathbb{D}$. We write $v[x \mapsto y]$ for the function equal to v, except that x is mapped to y.

Let $\mathbb{S}$ be the set of streams. Although satisfaction is defined over streams, a monitor will always receive only a finite stream prefix. We write $\mathbb{P}$ for the set of prefixes and $\preceq$ for the usual prefix order on streams and prefixes. For a prefix π and $i < |\pi|$, $\pi[i]$ denotes π's i-th element.

4 Slicing Framework

We introduce a general framework for parallel online monitoring based on slicing. Basin et al. [11] provide operators that split finite logs offline into independently monitorable slices, based on the events' data values and time-stamps. Each slice contains only a subset of the events from the original trace, which reduces the computational effort required to monitor the slice. We adapt this idea to online monitoring. Our framework is abstract. We start with a characterization of an online monitor's input–output behavior (Section 4.1). Slicing's fundamental property is that it preserves this behavior (Section 4.2). We then refine the framework and focus on the data in the events, since slicing with respect to time is more suitable for offline monitoring (Section 4.3).

4.1 Monitor Functions

Abstractly, a *monitor function* $\mathcal{M} \in \mathbb{P} \to \mathbb{O}$ maps stream prefixes to verdict outputs from some set $\mathbb{O}$. A monitor is an algorithm that implements a monitor function. An *online* monitor receives incremental updates of a stream prefix and computes the corresponding verdicts. We consider time-stamped databases to be the atomic units of the online monitor's input. The monitor may produce the verdicts incrementally, too. To represent this behavior at the level of monitor functions, we assume that verdict outputs are equipped with a partial order $\sqsubseteq$, where $o_1 \sqsubseteq o_2$ means that o_2 provides more (or the same) information as o_1. We also assume that $\mathcal{M}$ is a monotone map from the poset $\langle \mathbb{P}, \preceq \rangle$, i.e., stream prefixes ordered by the prefix relation, to the poset $\langle \mathbb{O}, \sqsubseteq \rangle$. This captures the intuition that as the monitor function receives more input, it produces more output, and, depending on the partial order $\sqsubseteq$, it does not retract previous verdicts.

The standard application of monitors for runtime verification is detecting violations of a safety property of the form $\Box \forall x_1 \dots x_n. \varphi$. To do this, one can monitor the negation $\neg\varphi$ to obtain the valuations of the variables $x_1, \dots, x_n$ that satisfy the negation. Such valuations correspond to the violations of the initial safety property. We call monitors that output valuations of the free variables *informative*.

Intuitively, the verdict of an informative monitor function $\mathcal{M}_\varphi$ is a set of tuples (v, i), where v is a valuation of the free variables of the MFOTL formula φ and i is an index in the event stream. We call these tuples *satisfying valuations*. Thus, we instantiate $\langle \mathbb{O}, \sqsubseteq \rangle$ with $\langle (V(\varphi) \to \mathbb{D}) \times \mathbb{N}, \subseteq \rangle$ when we work with an informative monitor function. By using the subset relation as the partial order on verdicts, the granularity at which an online implementation can incrementally output its verdict is at the level of satisfying valuations. The following definition makes the above intuition more precise.

Definition 1 An *informative monitor function* $\mathcal{M}_\varphi$ for φ is a monotone function $\langle \mathbb{P}, \preceq \rangle \to \langle (V(\varphi) \to \mathbb{D}) \times \mathbb{N}, \subseteq \rangle$ satisfying

Soundness	$\forall \pi, v, i.\ (v, i) \in \mathcal{M}_\varphi(\pi) \implies \forall \rho \succeq \pi.\ v, i \models_\rho \varphi$
Completeness	$\forall \pi, \rho, v, i.\ \pi \preceq \rho \wedge (\forall \rho' \succeq \pi.\ v, i \models_{\rho'} \varphi) \implies \exists \pi' \preceq \rho.\ (v, i) \in \mathcal{M}_\varphi(\pi')$

Soundness restricts the output to valuations that are satisfied independently of future events: the monitor may output a tuple (v, i) only if it is a satisfying valuation for all streams ρ extending the prefix π. This property is sometimes called *impartiality* [41]. Our definition of completeness is a weak form of *anticipation* [41]: once a valuation v is satisfied at an index i on every possible extension of the prefix π, the monitor must eventually output this fact. However, we allow the output to be delayed, which is generally necessary for formulas with future modalities. The delay may be unbounded with respect to either time or the number of databases alone. We therefore require that for any choice of the infinite stream extension $\rho \succeq \pi$, there is another prefix $\pi' \preceq \rho$ such that $\mathcal{M}_\varphi(\pi')$ contains the satisfying valuation (v, i). Informative monitor functions are not unique because the output delay is not fixed.

As concrete examples, the MonPoly monitor [16] implements an informative monitor function for a practically relevant fragment of MFOTL [15]. MonPoly's output delay depends only on the future operators' intervals in the monitored formula. The DejaVu monitor [37] internally computes an informative monitor function for a past-only fragment of MFOTL, where all intervals are $[0, \infty)$. It represents valuations as binary decision diagrams (BDDs), but does not output them. Instead, DejaVu's verdicts consist only of the

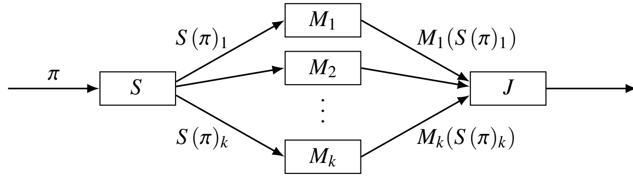


Fig. 2 The parallelized monitor function $J\big(\lambda k.\ M_k(S(\pi)_k)\big)$, assuming $K=\{1,\dots,k\}$

indices where violations occurred. Since DejaVu does not support future operators, its verdict output is never delayed.

We briefly compare our informative monitor functions with another common type of monitor functions from the literature where $\mathbb{O}$ is the set $\{?,\bot,\top\}$ and the partial order $\sqsubseteq$ is the reflexive closure of $\{(?,\bot),(?,\top)\}$ [19, 46]. The verdict $\bot$ means that the monitored prefix is a bad prefix, i.e., all its infinite extensions violate the formula. Similarly, $\top$ denotes a good prefix, while ? indicates an inconclusive result. Every nonempty result from $\mathcal{M}_\varphi(\pi)$ corresponds to a $\bot$ verdict for the formula $\Box\forall x_1\dots x_n.\neg\varphi$ (due to soundness), whereas an empty result could either mean ? or $\top$.

4.2 Abstract Slicing

Parallelizing a monitor should not affect its input–output behavior. We formulate this correctness requirement abstractly using the notion of a slicer for a monitor function. The slicer specifies how to split a stream prefix into independently monitorable substreams, called slices, and how to combine the verdict outputs of the parallel submonitors into a single verdict.

Definition 2 A *slicer* for a monitor function $\mathcal{M}\in\mathbb{P}\to\mathbb{O}$ is a tuple (K,M,S,J), where K is a set of slice identifiers, the *submonitor* family $M\in K\to(\mathbb{P}\to\mathbb{O})$ is a K-indexed family of monitor functions, the *splitter* $S\in\mathbb{P}\to(K\to\mathbb{P})$ splits prefixes into K-indexed slices, and the *joiner* $J\in(K\to\mathbb{O})\to\mathbb{O}$ combines K-indexed verdicts into a single one, satisfying:

Monotonicity $\forall\pi_1,\pi_2,k.\ \pi_1\preceq\pi_2\Longrightarrow S(\pi_1)_k\preceq S(\pi_2)_k$.
Correctness $\forall\pi.\ J\big(\lambda k.\ M_k(S(\pi)_k)\big)=\mathcal{M}(\pi)$.

For an input prefix π, let $S(\pi)$ denote the collection of its slices. Each slice is identified by an element $k\in K$, which we write as a subscript. We require the splitter S to be monotone so that the submonitors M_k, which may differ from the monitor function $\mathcal{M}$, can process the sliced prefixes incrementally. Composing the splitter, the corresponding submonitor for each slice, and the joiner as shown in Figure 2 yields the parallelized monitor function $J\big(\lambda k.\ M_k(S(\pi)_k)\big)$. This function is correct if and only if it computes the same verdicts as $\mathcal{M}$.

For example, parametric trace slicing [49, 51] can be seen as a particular slicer for monitor functions that arise from sliceable FO-LTL formulas [49, Section 4]. Thereby, K is the Cartesian product of finite domains for the formulas' variables. The elements of K are thus valuations and the splitter is defined as the restriction of the trace to the values occurring in the valuation. The submonitor M_k is a propositional LTL monitor and the joiner simply takes the union of the results (which may be marked with the valuation).

The splitter S as defined above is overly general. A concrete instance of S may determine each event's assignment to slices based on all previous events. In practice, we would like an efficient implementation of S. For example, parametric trace slicing determines the target slice for an event by inspecting events individually (and not as part of the entire prefix). We call a splitter with this property event-separable. Event-separable splitters are desirable because they cater for a parallel implementation of the splitter itself.

Definition 3 A splitter S is called *event-separable* if there is a function $\hat{S}\in(\mathbb{E}\times\mathbb{D}^*)\to\mathcal{P}(K)$ such that $S(\pi)_k[i]=(\tau_i,\{e\in D_i\mid k\in\hat{S}(e)\})$, for all $\pi\in\mathbb{P}$, $k\in K$, and $i<|\pi|$.

Lemma 1 *Assume that S is event-separable. Then $\pi_1\preceq\pi_2$ implies $S(\pi_1)_k\preceq S(\pi_2)_k$ for all $k\in K$.*

Proof Fix an event-separable splitter S with the corresponding function $\hat{S}$ (*). Fix two prefixes $\pi_1=(\tau_i,D_i)_{i<|\pi_1|}$ and $\pi_2=(\tau'_i,D'_i)_{i<|\pi_2|}$, with $\pi_1\preceq\pi_2$. We thus have $\tau_i=\tau'_i$ and $D_i=D'_i$ for all $i<|\pi_1|$ (**). Fix $k\in K$. We show $S(\pi_1)_k\preceq S(\pi_2)_k$ pointwise by showing $S(\pi_1)_k[i]=S(\pi_2)_k[i]$ for all $i<|\pi_1|$. To do so we calculate:

$$S(\pi_1)_k[i]\overset{(*)}{=}(\tau_i,\{e\in D_i\mid k\in\hat{S}(e)\})$$
$$\overset{(**)}{=}(\tau'_i,\{e\in D'_i\mid k\in\hat{S}(e)\})\overset{(*)}{=}S(\pi_2)_k[i].\qquad\square$$

We also call a slicer with an event-separable splitter *event-separable*. We identify an event-separable slicer (K,M,S,J) with $(K,M,\hat{S},J)$, where $\hat{S}$ is the function from Definition 3.

4.3 Joint Data Slicer

We now describe an event-separable slicer for informative monitor functions $\mathcal{M}_\varphi$. Our *joint data slicer* distributes events according to the valuations they induce in the formula. Recall that the output of $\mathcal{M}_\varphi$ consists of all valuations that satisfy the formula φ at some index. For a given valuation, only a subset of the events is relevant to evaluate the formula. We would like to evaluate φ separately for each valuation to determine whether it is satisfied by that valuation, as this would allows us to exclude some events from each slice. However, there are infinitely many valuations in the presence of infinite domains. Therefore, the joint data slicer uses finitely many (possibly overlapping) slices associated with sets of valuations, which taken together cover all possible valuations.

We assume without loss of generality that the bound variables in φ are disjoint from the free variables $V(\varphi)$. Given

an event $e = r(d_1, \dots, d_{\iota(r)})$, the set $matches(\varphi, e)$ contains all valuations $v \in V(\varphi) \to \mathbb{D}$ for which there is a subformula $r(t_1, \dots, t_{\iota(r)})$ in φ where $v(t_i) = d_i$ for all $i \in \{1, \dots, \iota(r)\}$. Intuitively, v is in $matches(\varphi, e)$ if the event e is possibly relevant for evaluating φ over the valuation v.

Definition 4 Let φ be an MFOTL formula and $f \in (V(\varphi) \to \mathbb{D}) \to \mathcal{P}(K)$ be a mapping from valuations to nonempty sets of slice identifiers. The *joint data slicer* for φ with splitting strategy f is the tuple $(K, \lambda k.\ \mathcal{M}_\varphi, \hat{S}_f, J_f)$, where[1]

$$\hat{S}_f(e) = \bigcup_{v \in matches(\varphi,e)} f(v),$$

$$J_f(s) = \bigcup_{k \in K} (s_k \cap (\{v \mid k \in f(v)\} \times \mathbb{N})).$$

The splitting strategy f associates valuations to slices (more precisely, slice identifiers). Accordingly, $\hat{S}_f$ assigns the event e to all slices k for which there exists $v \in matches(\varphi, e)$, i.e., a valuation v for which e may be relevant, with $k \in f(v)$. The joiner J_f takes the union of the verdicts from all slices, keeping only those verdicts that the corresponding slice is responsible for. Note that $\{v \mid k \in f(v)\} \times \mathbb{N}$ is the set of all verdicts whose valuation is associated with the slice k.

The following example demonstrates why the intersection in the definition of J_f is needed for some formulas, for example those involving equality. Intuitively, these formulas may be satisfied if and only if certain events are absent. The problem occurs if the input prefix contains these events, but a slice does not.

Example 1 Consider the formula $\varphi = x \approx a \land \neg P(x)$, where a is a constant, and consider a stream ρ with the prefix $\pi = \langle (0, \{P(a)\}) \rangle$. Obviously, $v, 0 \not\models_\rho \varphi$ for all v. However, the event $P(a)$ will be omitted from each slice k that does not have an associated valuation mapping x to a. (A splitting strategy with such a slice exists whenever $|K| \geq 2$.) Hence $v[x \mapsto a], 0 \models_{\rho'} \varphi$ for all v and all extensions ρ' of the slice $S_f(\pi)_k$ to a stream. The result will be unsound if we do not filter the erroneous satisfying valuations $v[x \mapsto a]$ that are necessarily output by the k-th submonitor (due to its completeness).

We show next that $\mathcal{M}^f_\varphi(\pi) = J_f\big(\lambda k.\ \mathcal{M}_\varphi(S_f(\pi)_k)\big)$, the parallelized monitor that uses the joint data slicer, is an informative monitor function, i.e., it is monotone, sound, and complete. As a first step, given a formula φ and a set of valuations R, we define the formula's *relevant events* with respect to R as $E_\varphi(R) = \{e \mid R \cap matches(\varphi, e) \neq \{\}\}$. The following lemma justifies this name: if we restrict the databases in a stream to (a superset of) the formula's relevant events with respect to R, the satisfying valuations within R remain unchanged.

Lemma 2 *Fix a formula φ, a stream $\rho = (\tau_i, D_i)_{i \in \mathbb{N}}$, a set of valuations R, and a set of events E, with $E_\varphi(R) \subseteq E$. Let $\sigma = (\tau_i, D_i \cap E)_{i \in \mathbb{N}}$. Then $v, i \models_\rho \varphi \Longleftrightarrow v, i \models_\sigma \varphi$ for all $v \in R$ and $i \in \mathbb{N}$.*

[1] Recall that s is a family of K-indexed verdicts, so s_k denotes the verdict for slice k.

Proof Proof by structural induction over the formula φ, generalizing over v, R, and i. We only show the base cases, which are the most interesting ones, and the step case for $\exists$. The other step cases all follow easily from the induction hypothesis because the evaluation only depends on the evaluation of the recursive subformulas (covered by the induction hypothesis) and the time-stamps in the streams. Note that the latter are the same in ρ and σ.

Case $\varphi = r(t_1, \dots, t_n)$ with $n = \iota(r)$: We have for any $v \in R$:

$$v, i \models_\rho \varphi \Longleftrightarrow r(v(t_1), \dots, v(t_n)) \in D_i \overset{*}{\Longleftrightarrow} r(v(t_1), \dots, v(t_n)) \in D_i \cap E \Longleftrightarrow v, i \models_\sigma \varphi.$$

The step marked with $*$ is justified as follows. Either $r(v(t_1), \dots, v(t_n)) \notin D_i$, and both sides of $\Longleftrightarrow$ are false. Otherwise, $r(v(t_1), \dots, v(t_n)) \in D_i$, which implies that $v \in matches(\varphi, (r(v(t_1), \dots, v(t_n)))$. This in turn implies that $r(v(t_1), \dots, v(t_n)) \in E_\varphi(R) \subseteq E$ using the fact that $v \in R$ and the lemma's assumption.

Case $\varphi = t_1 \approx t_2$: We have for any $v \in R$: $v, i \models_\rho \varphi \Longleftrightarrow v(t_1) = v(t_2) \Longleftrightarrow v, i \models_\sigma \varphi$.

Case $\varphi = \exists x.\ \psi$: We have for any $v \in R$:

$$v, i \models_\rho \varphi \Longleftrightarrow \exists z \in \mathbb{D}.\ v[x \mapsto z], i \models_\rho \psi \overset{*}{\Longleftrightarrow} \exists z \in \mathbb{D}.\ v[x \mapsto z], i \models_\sigma \psi \Longleftrightarrow v, i \models_\sigma \varphi.$$

The step marked with $*$ is justified using the induction hypothesis for the formula ψ, namely, $v[x \mapsto z], i \models_\rho \varphi \Longleftrightarrow v[x \mapsto z], i \models_\sigma \varphi$, for all $z \in \mathbb{D}$. Note that we have instantiated the parameters v and R by $v[x \mapsto z]$ and $\{v[x \mapsto z] \mid v \in R\}$, respectively. ☐

The relevant events provide an alternative characterization of the joint data slicer's splitter: $S_f(\pi)_k[i] = (\tau_i, D_i \cap E_\varphi(\{v \mid k \in f(v)\}))$, for all $\pi = (\tau_i, D_i)_{i<|\pi|}$ and $i < |\pi|$.

Theorem 1 *The function $\mathcal{M}^f_\varphi(\pi) = J_f\big(\lambda k.\ \mathcal{M}_\varphi(S_f(\pi)_k)\big)$ is an informative monitor function.*

Proof The monotonicity of $\mathcal{M}^f_\varphi$ follows directly from $\mathcal{M}_\varphi$'s monotonicity. For soundness, fix i, v, and π and assume $(v, i) \in \mathcal{M}^f_\varphi(\pi)$. Then, by $\mathcal{M}^f_\varphi$'s definition, we obtain a slice identifier $k \in f(v)$ such that $(v, i) \in \mathcal{M}_\varphi(S_f(\pi)_k)$. From $\mathcal{M}_\varphi$'s soundness, we have $v, i \models_\sigma \varphi$ for all $\sigma \succeq S_f(\pi)_k$. Let ρ be some stream extending π, i.e., $\rho \succeq \pi$. Using the alternative characterization of S_f and Lemma 2 with R instantiated to $\{v \mid k \in f(v)\}$, we deduce $v, i \models_\rho \varphi$.

For completeness, fix i, v, π, and $\rho = (\tau_i, D_i)_{i \in \mathbb{N}}$, where $\pi \preceq \rho$ (i.e., $\pi = (\tau_i, D_i)_{i<|\pi|}$), and $\forall \rho' \succeq \pi.\ v, i \models_{\rho'} \varphi$. As $f(v)$ is nonempty, let $k \in f(v)$ be some slice identifier. We first show that for all $\sigma = (\tau'_i, D'_i)_{i \in \mathbb{N}}$ with $\sigma \succeq S_f(\pi)_k$ we have $v, i \models_\sigma \varphi$. For $i < |\pi|$, we have $\tau'_i = \tau_i$ and $D'_i = D_i \cap E_\varphi(\{v \mid k \in f(v)\})$ via S_f's alternative characterization. Let the stream $\rho' = (\tau'_i, E_i)_{i \in \mathbb{N}}$, where $E_i = D_i$ for $i < |\pi|$ and

$E_i = D'_i$ otherwise, and let the stream $\sigma' = (\tau'_i, E_i \cap E_\varphi(\{v \mid k \in f(v)\}))_{i\in\mathbb{N}}$. We calculate using Lemma 2 with $R = \{v \mid k \in f(v)\}$:

$$v,i \models_\sigma \varphi \overset{\text{Lemma 2}}{\iff} v,i \models_{\sigma'} \varphi \overset{\text{Lemma 2}}{\iff} v,i \models_{\rho'} \varphi.$$

Because $\rho' \succeq \pi$, we have $v,i \models_{\rho'} \varphi$ by our assumption, and thus $v,i \models_\sigma \varphi$. Now, we can apply $\mathcal{M}_\varphi$'s completeness to the stream $\sigma'' = (\tau_i, D_i \cap E_\varphi(\{v \mid k \in f(v)\})_{i\in\mathbb{N}}$, to obtain a π'' such that $\pi'' \preceq \sigma''$ and $(v,i) \in \mathcal{M}_\varphi(\pi'')$. Taking $\pi' = (\tau_i, D_i)_{i<|\pi''|}$, we have $\pi' \preceq \rho$ and $\pi'' = S_f(\pi')_k$. By the definition of $\mathcal{M}^f_\varphi$, we conclude that $(v,i) \in \mathcal{M}^f_\varphi(\pi')$. □

The monitor functions $\mathcal{M}_\varphi$ and $\mathcal{M}^f_\varphi$ may differ. However, both are informative, i.e., they produce correct verdicts (and eventually all verdicts by completeness) for the formula φ. Yet they may output verdicts with different delays. In general, the joint data slicer is only a slicer for $\mathcal{M}^f_\varphi$ but not for $\mathcal{M}_\varphi$.

Corollary 1 *The joint data slicer $(K, \lambda k.\ \mathcal{M}_\varphi, \hat{S}_f, J_f)$ is a slicer for $\mathcal{M}^f_\varphi$.*

Proof Monotonicity follows from Lemma 1; correctness follows from $\mathcal{M}^f_\varphi$'s definition. □

The joint data slicer is also a slicer for the original monitor function $\mathcal{M}_\varphi$, i.e., it produces the same output as the original monitor function, under an additional assumption on $\mathcal{M}_\varphi$.

Definition 5 A monitor function is *sliceable* if for any prefix $\pi = (\tau_i, D_i)_{i<|\pi|}$, set of valuations R, and $v \in R$, we have $(v,i) \in \mathcal{M}_\varphi((\tau_i, D_i \cap E_\varphi(R))_{i<|\pi|}) \iff (v,i) \in \mathcal{M}_\varphi(\pi)$.

This assumption is satisfied by MonPoly's and DejaVu's concrete monitor functions: The indices at which these monitors output satisfying valuations depend only on the sequence of time-stamps, which slicing does not affect. It follows from Lemma 2 that they are sliceable.

Theorem 2 *The joint data slicer $(K, \lambda k.\ \mathcal{M}_\varphi, \hat{S}_f, J_f)$ is a slicer for $\mathcal{M}_\varphi$, if $\mathcal{M}_\varphi$ is sliceable.*

Proof Monotonicity follows from Lemma 1. For correctness, we must show that $(v,i) \in \mathcal{M}^f_\varphi(\pi)$ if and only if $(v,i) \in \mathcal{M}_\varphi(\pi)$, for an arbitrary v and i. This follows from the definition of $\mathcal{M}^f_\varphi$, the sliceability assumption, and that $f(v)$ is nonempty. □

Example 2 Consider the formula $P(x,y) \wedge \neg \Diamond_{[0,5]} (P(y,x) \vee Q(x,y))$. We apply the joint data slicer with $K = \{1,2\}$ and a splitting strategy f that maps the valuation $\langle x = 5, y = 7\rangle$ to the first slice and all other valuations to the second slice. We obtain the following slices for the prefix $\pi = \langle(11, \{P(7,5)\}),\ (12, \{P(5,1), Q(7,5)\}),\ (21, \{P(5,7), Q(5,7)\})\rangle$:

$$S_f(\pi)_1 = \langle(11,\{P(7,5)\}),\ (12,\{\}),\ (21,\{P(5,7),Q(5,7)\})\rangle$$
$$S_f(\pi)_2 = \langle(11,\{P(7,5)\}),\ (12,\{P(5,1),Q(7,5)\}),\ (21,\{P(5,7)\})\rangle.$$

The events $P(5,7)$ and $P(7,5)$ are duplicated across the slices because both $\langle x = 5, y = 7\rangle$ and $\langle x = 7, y = 5\rangle$ are matching valuations for either event. The joiner is crucial for the slicer's correctness in this example. Because of the subformula $P(y,x)$, the first slice receives the event $P(7,5)$ but not the event $Q(7,5)$, which is sent to the second slice instead. This results in the spurious verdict $\langle x = 7, y = 5\rangle$ at index 0, which the joiner's intersection filters out.

The data slicer used in the offline slicing framework [11] is defined for a single free variable x and a collection $(S_k)_{k\in K}$ of slicing sets covering the domain: $\bigcup_{k\in K} S_k = \mathbb{D}$. This single variable slicer is a special case of our joint data slicer. To see this, define $f(v)$ to be the set of all k with $v(x) \in S_k$. At least one such k must exist because the S_k cover the domain. In contrast, some instances of the joint data slicer cannot be simulated by composing single variable slicers. This limitation affects formulas where the same predicate symbol appears in multiple atoms that each miss at least one free variable to slice on. As a result, single variable slicers are ineffective for some formulas as they add unnecessary data duplication.

Example 3 Consider the formula $P(x) \wedge \bullet P(y)$ and the splitting strategy that maps v to the slice $(v(x) \bmod 2, v(y) \bmod 2)$ such that there are four slices in total. Any single variable slicer will send each P event to all slices, and this extends to their composition. The joint data slicer sends each event $P(d)$ to exactly three slices, excluding the slice (z,z), where $(z \bmod 2) \neq (d \bmod 2)$. This example generalizes to other splitting strategies as we show in Example 7 in Section 5.3.

Finally, we revisit the intersection with $\{v \mid k \in f(v)\} \times \mathbb{N}$ in the definition of J_f. Examples 1 and 2 demonstrate the need for it in general. A valid question is for which formulas and splitting strategies can the intersection be omitted, i.e., when can we replace J_f with $J'(s) = \bigcup_{k\in K} s_k$? For example, this replacement is necessary when using DejaVu as a submonitor (see Section 6). We give a sufficient condition stemming from the following lemma. The lemma ensures that a formula's satisfying valuations on streams restricted to relevant events with respect to a given set of valuations R come from precisely this set of valuations R.

Lemma 3 *Let φ be a formula and $R \neq \{\}$ a nonempty set of valuations. Assume that*

(1) the formula φ is safe*, i.e., satisfies the predicate* sf *defined in Figure 3;*
(2) no subformula of the form $x \approx a$ or $a \approx x$, where $x \in \mathbb{V}$ and $a \in \mathbb{D}$, occurs in φ;
(3) no event name occurs twice in φ; and
(4) for all $v_1 \in R$, $v_2 \in R$, and v satisfying $v(x) = v_1(x) \vee v(x) = v_2(x)$ for all $x \in V(\varphi)$, we have $v \in R$.

If $v,i \models_\rho \varphi$ for some $i \in \mathbb{N}$, a valuation v, and a stream $\rho = (\tau_i, D_i)_{i\in\mathbb{N}}$ with $D_i \subseteq E_\varphi(R)$ for all $i \in \mathbb{N}$, then $v \in R$.

$$
\begin{aligned}
\mathsf{sf}(r(t_1,\dots,t_{\iota(r)})) &= \mathsf{true}\\
\mathsf{sf}(t_1 \approx t_2) &= (\exists a \in \mathbb{D}.\, t_1 = a \lor t_2 = a)\\
\mathsf{sf}(\neg(t_1 \approx t_2)) &= (\exists a, b \in \mathbb{D}.\, t_1 = a \land t_2 = b) \lor\\
&\quad\ (\exists x \in \mathbb{V}.\, t_1 = x \land t_2 = x)\\
\mathsf{sf}(\neg(\varphi \lor (\neg\psi))) &= ((\mathsf{sf}(\varphi) \land V(\varphi) \subseteq V(\psi)) \lor \mathsf{sf}^{\neg}(\varphi)) \land \mathsf{sf}(\psi)\\
\mathsf{sf}(\varphi \lor \psi) &= \mathsf{sf}(\varphi) \land \mathsf{sf}(\psi) \land V(\varphi) = V(\psi)\\
\mathsf{sf}(\exists x.\, \varphi) &= \mathsf{sf}(\varphi)\\
\mathsf{sf}(\CIRCLE_I\, \varphi) &= \mathsf{sf}(\varphi)\\
\mathsf{sf}(\bigcirc_I\, \varphi) &= \mathsf{sf}(\varphi)\\
\mathsf{sf}(\varphi \,\mathsf{S}_I\, \psi) &= V(\varphi) \subseteq V(\psi) \land (\mathsf{sf}(\varphi) \lor \mathsf{sf}^{\neg}(\varphi)) \land \mathsf{sf}(\psi)\\
\mathsf{sf}(\varphi \,\mathsf{U}_I\, \psi) &= V(\varphi) \subseteq V(\psi) \land (\mathsf{sf}(\varphi) \lor \mathsf{sf}^{\neg}(\varphi)) \land \mathsf{sf}(\psi)
\end{aligned}
$$

Fig. 3 Safe formulas ($\mathsf{sf}^{\neg}(\varphi)$ abbreviates $\mathsf{sf}(\psi)$ if $\varphi = \neg\psi$ and false otherwise)

Proof (Sketch) By induction on the structure of safe formulas. The base cases are straightforward using the assumptions *(2)* and *(3)*. Note that safe formulas only allow negation to occur in formulas of the form $(\neg\varphi) \land \psi$ (i.e., $\neg(\varphi \lor (\neg\psi))$), $(\neg\varphi)\,\mathsf{S}_I\,\psi$, and $(\neg\varphi)\,\mathsf{U}_I\,\psi$ with all the free variables of the negated subformula $\neg\varphi$ being contained in the free variables of ψ. This ensures that the satisfying valuations of these formulas are a subset of the satisfying valuations of ψ, allowing for a straightforward use of the induction hypothesis. The case $\varphi \land \psi$ (where both subformulas are not negated), requires joining the satisfying valuations of φ and ψ. Condition *(4)* makes sure that this join operation produces a valuation in R. □

The safety assumption requires that any negated subformula is guarded by a non-negated subformula, such that φ can be monitored using finite relations [15, 55]. (Safe formulas are called *monitorable* in these references.) The safety assumption is standard for monitors operating on finite tables. For instance, the MonPoly monitor only supports safe formulas [15]. In contrast, DejaVu supports unsafe formulas for the past-only non-metric fragment of MFOTL [37]. Observe that condition *(2)* of Lemma 3 rules out the formula from Example 1. We conclude this section with J''s main property.

Theorem 3 *Let φ be a formula and f the joint data slicer's splitting strategy. Let $R(k) = \{v \mid k \in f(v)\}$ and assume that f makes $R(k)$ nonempty for all $k \in K$. Under the assumptions (1)–(3) of Lemma 3 on φ and assumption (4) on $R(k)$ for all $k \in K$, we have:*

$$J_f\big(\lambda k.\, \mathcal{M}_\varphi(S_f(\pi)_k)\big) = J'\big(\lambda k.\, \mathcal{M}_\varphi(S_f(\pi)_k)\big).$$

Proof The left-to-right inclusion is obvious. For the right-to-left inclusion, assume $(v,i) \in J'\big(\lambda k.\, \mathcal{M}_\varphi(S_f(\pi)_k)\big)$. Then, obtain $k \in K$ such that $(v,i) \in \mathcal{M}_\varphi(S_f(\pi)_k)$. By the monitor function $\mathcal{M}_\varphi$'s soundness, we have $v, i \models_\rho \varphi$ for all $\rho \succeq S_f(\pi)_k$. Taking any $\rho = (\tau_i, D_i)_{i\in\mathbb{N}}$ satisfying $D_i \subseteq E_\varphi(R(k))$ for all i (note that this is precisely what $\rho \succeq S_f(\pi)_k$ ensures for $i < |\pi|$) and applying Lemma 3, we have $v \in R(k)$ and thus $(v,i) \in J_f\big(\lambda k.\, \mathcal{M}_\varphi(S_f(\pi)_k)\big)$. □

5 Automatic Slicing

The joint data slicer is parameterized by a splitting strategy. Ideally, the chosen strategy optimally utilizes the available computing resources: computation costs should be evenly distributed and any overhead kept low. In this section, we present our approach to automatically selecting a suitable strategy. It is inspired by results from database theory and leverages stream statistics to optimize the submonitors' event rates, i.e., the number of events in a time period.

In online monitoring, the monitor's throughput must be high enough to process the incoming events with bounded delay, especially if its buffering capacity is limited. The goal of slicing is to supply the submonitors with substreams that can be monitored more efficiently than the entire event stream. Under the assumption that slicing and the communication to the submonitors do not pose a bottleneck, the parallel monitor will thus achieve a higher throughput than the sequential monitor. Another related benefit is the improved worst-case latency in the presence of bursty event streams, where the events are not distributed evenly in time. Low latency is important in online monitoring to obtain timely verdicts.

The key problem we solve is to find a splitting strategy that achieves the above goal. Ideally, the improvements in throughput and worst-case latency scale with the number of submonitors. To approximate this ideal within our slicing framework, the splitting strategy should minimize the event rates observed by a fixed number of submonitors. This in turn maximizes the parallel monitor's throughput if we make the simplifying assumption that the submonitors' throughput solely depends on their input event rate. Under the same assumption, the submonitors require less memory. We do not optimize the communication cost in this article. However, the number of slices is a parameter that affects the communication cost due to data duplication.

5.1 Recap of the Hypercube Algorithm

Our automatic splitting strategy is based on the observation that the hypercube algorithm [4, 33, 57], which is used to parallelize relational queries in databases, can be generalized to the online monitoring of MFOTL formulas.

We start by recalling the standard notion of full conjunctive queries [2], which represent a substantially less expressive language than MFOTL. The computational properties of conjunctive queries are well understood. In particular, researchers have devised and analyzed (near-)optimal distributed algorithms for computing conjunctive queries [3, 4, 20, 21, 38, 39]. Afterwards, we focus on the hypercube algorithm and recall previous results. The terminology we use has been adjusted slightly to match the monitoring setting.

A *database instance* (or *database* for short) represents a finite set of events. This coincides with the definition that

we previously gave for the stream elements in MFOTL's semantics. In the database context, we also call the names $r \in \mathbb{E}$ *relation names*. A *relation* $D(r)$ in a database D is the set of all events in D with the name r. Its *size* $|D(r)|$ is the cardinality of the set $D(r)$. The *degree* of a value $d \in \mathbb{D}$ with respect to an attribute $i \in \{1,\dots,\iota(r)\}$ of the relation name r is the number of events $r(d_1,\dots,d_{\iota(r)}) \in D$ with $d_i = d$.

A *query* q is a syntactic expression in a given query language. It defines a mapping $q(D)$ from databases D to finite sets of valuations over some finite set of variables $V(q) \subset \mathbb{V}$. An *atom* is an expression $r(y_1,\dots,y_{\iota(r)})$, where $r \in \mathbb{E}$, and the variables y_i are elements of $\mathbb{V}$. The *image* of an atom $a = r(y_1,\dots,y_{\iota(r)})$ under a valuation v is the event $v(a) = r(v(y_1),\dots,v(y_{\iota(r)}))$. We write $V(a)$ for the set of variables $\{y_1,\dots,y_{\iota(r)}\}$. A *full conjunctive query* q is a finite set of atoms. Such a query maps to valuations that have as their domain the variables occurring in the query's atoms, i.e., $V(q) = \bigcup_{a\in q} V(a)$. The semantics of q is then given by

$$q(D) = \{v \in V(q) \to \mathbb{D} \mid \forall a \in q.\ v(a) \in D\}.$$

Note that we overload q above and refer to it both as a set (denoting the query's syntax) and as a mapping (denoting the query's semantics). In the following, we assume that there is a linear ordering $x_1,\dots,x_n$ on the variables $V(q)$.

The basic hypercube algorithm [4] computes a full conjunctive query q on a distributed, MapReduce-like system [29] with p parallel workers. The algorithm is parametrized by the number of workers p and by the *share* $p_i \in \mathbb{N}$ for each variable x_i in q, where $i \in \{1,\dots,n\}$. Each worker is assigned a unique coordinate vector $(x_1,\dots,x_n) \in [p_1] \times \cdots \times [p_n]$, where $[p] := \{0,\dots,p-1\}$ for $p \in \mathbb{N}$. Initially, the events of the database D are assumed to be distributed evenly (but in an unspecified manner) over the p workers. The algorithm proceeds as follows.

1. Hash functions $h_i \in \mathbb{D} \to [p_i]$ are chosen randomly and independently for all $i \in \{1,\dots,n\}$. The hash functions are known to all workers.
2. In the *map* phase, each worker computes for every event e in its local partition a set of workers $T(e)$, represented by their coordinates, to which it sends the event:
$$T(e) = \bigcup_{v\in matches(q,e)} \{(h_1(v(x_1)),\dots,h_n(v(x_n)))\}, \quad (1)$$
where $matches(q,e) = \{v \mid \exists a \in q.\ v(a) = e\}$. The set $T(e)$ can be computed by determining for every $a \in q$ the unique $v_a \in V(a) \to \mathbb{D}$ with $v_a(a) = e$ (if it exists). Each partial valuation v_a fixes the coordinates $h_i(v_a(x_i))$ for all $x_i \in V(a)$, while for the remaining coordinate components $x_i \notin V(a)$, every possible combination of the coordinates in $[p_i]$ must be considered. (We assume for simplicity that the hash functions are surjective.)
3. In the *reduce* phase, the workers evaluate q locally on the events that they received in the first phase. The query's result is the union of all local results, which may optionally be sent to a centralized worker.

In general, the basic hypercube algorithm duplicates events, namely those matching an atom that does not contain a variable x_i with $p_i > 1$. The total number of events that each worker receives (and on which it computes q) depends on the input database and the shares. Beame et al. [21] analyze the maximum worst-case *load* of the workers, given fixed relation sizes and shares. They define the load as the total size of the messages (in bits) received by a worker before the algorithm's reduce phase. Based on the number of workers p, Beame et al. distinguish between *skewed* and *skew-free* databases. A database is skewed if it contains *heavy hitters*, which are values whose degree with respect to some attribute i and relation name r exceeds $|D(r)|/p$. For skew-free databases, they show that the maximum load generated by the hypercube algorithm is asymptotically bounded (up to a factor polylogarithmic in p) by $L = \sum_{a\in q} |D(r)|/\left(\prod_{x_i\in V(a)} p_i\right)$ with high probability. (In fact, they prove the bound for a more general notion of skew-free databases.)

Beame et al. [21] also show that the shares can be optimized using linear programming. The input to the optimization is the full conjunctive query and the relation sizes of the database on which the query should be computed. Using a single round of communication and the optimized shares, the hypercube algorithm matches the lower bound for the maximum load that is necessary to compute the query. A single round of communication means that only one communication step is allowed after the initial communication phase. Afterwards, each worker can only perform local computations. The lower bound holds under the assumption that p workers can send arbitrary messages over private channels, have unbounded computational power, and have access to a common source of randomness.

However, optimizing the shares with linear programming does not yield integer values in general. As an alternative, Chu et al. [27] propose a simple exhaustive search over all possible integer shares, selecting the shares that minimize L. We present a modified version of their algorithm in Section 5.3 (Algorithm 2).

Example 4 (adapted from [21]) Consider the *star* query $q_S = \{P(x_1,x_2), Q(x_1,x_3), R(x_1,x_4)\}$ and a skew-free database D with $|D(P)| = |D(Q)| = |D(R)| = m$. The optimal shares for the hypercube algorithm with p workers are $p_1 = p$ and $p_2 = p_3 = 1$. This results in each worker receiving approximately $3m/p$ events. For the *triangle* query $q_T = \{P(x_1,x_2), Q(x_2,x_3), R(x_3,x_1)\}$ on the same database as before, the optimal shares are $p_1 = p_2 = p_3 = p^{1/3}$. Thus each worker receives approximately $3m/p^{2/3}$ events. If p is not a cubic number, we must approximate $p^{1/3}$ by a combination of

integers. E.g. for $p = 16$, the algorithm by Chu et al. selects $p_1 = 4$ and $p_2 = p_3 = 2$ (or a permutation of these numbers).

Next, we show how the events are distributed to the workers for q_T. We assume $p = 64$ (hence $p_1 = p_2 = p_3 = 4$) and simplify the hash functions to $h(x) = x \bmod 4$ for the purpose of this example. The slices are thus identified by three coordinates between 0 and 3, with one coordinate for each variable x_1, x_2, and x_3.

event e	target slices $T(e)$	event e	target slices $T(e)$
$P(0,1)$	010, 011, 012, 013	$Q(1,7)$	013, 113, 213, 313
$P(1,1)$	110, 111, 112, 113	$R(7,0)$	003, 013, 023, 033

The events $P(0,1)$, $Q(1,7)$, and $R(7,0)$ are sent to the worker with coordinates 013. This ensures that the valuation $\langle x_1 = 0, x_2 = 1, x_3 = 7\rangle \in q_T(D)$ is produced by at least this worker.

When the database is skewed, i.e., it contains heavy hitters, the basic hypercube algorithm sketched above is not optimal: applying a hash function h_i with share $p_i > 1$ to a heavy hitter does not distribute the value evenly over the coordinates $[p_i]$. Koutris et al. [39] propose an extension, which we simply call *the* hypercube algorithm, that is worst-case optimal also for skewed databases. Our automatic splitting strategy adjusts this algorithm to the online monitoring setting (Algorithm 1 in Section 5.3). Koutris et al. assume that all heavy hitters in the database are known in addition to the relation sizes. In the database setting, as well as in offline monitoring, this is a reasonable assumption since computing statistics is asymptotically dominated by querying.

In the hypercube algorithm, a copy of the basic algorithm is executed in parallel for every subset $H \subseteq V(q)$ of the query's variables. Each copy uses its own set of shares $p_{H,i}$ and hash functions $h_{H,i}$, but with the constraint that $p_{H,i} = 1$ if $x_i \in H$. A valuation v is *heavy in variable x* if there exists an atom $a = r(y_1,\dots,y_{\iota(r)}) \in q$ and i where $y_i = x$ and $v(x)$ is a heavy hitter in the attribute i of r. We write $\mathit{heavy}(q,v)$ for the set of variables in which v is heavy. The event e is processed by those instances of the basic hypercube algorithm that are associated with the variable sets $\mathit{heavy}(q,v)$ for which there exists an atom $a \in q$ with $v(a) = e$. For every H, the corresponding shares can be optimized as in the basic hypercube algorithm by considering the *residual query* q_H, which is obtained from q by removing all occurrences of variables in H.

Example 5 Suppose that the database D from Example 4 is skewed. We analyze the optimal shares for the triangle query q_T and instances of the basic hypercube algorithm for the variable subsets $H_1 = \{\}$, $H_2 = \{x_1\}$, $H_3 = \{x_1, x_2\}$, and $H_4 = \{x_1, x_2, x_3\}$. If no variable has a heavy hitter (H_1), the shares from Example 4 apply. The remaining variable sets have symmetric solutions. For the algorithm instance H_2, the optimal shares are $p_{H_2,1} = 1$ and $p_{H_2,2} = p_{H_2,3} = p^{1/2}$. Each worker then receives at most $1/p^{1/2}$ of the events for which only x is assigned a heavy hitter. For the algorithm instance H_3, the optimal shares are $p_{H_3,1} = p_{H_3,2} = 1$ and $p_{H_3,3} = p$, so at most $1/p$ of the corresponding events are sent to the workers. Finally for the algorithm instance H_4, one must broadcast the events to all workers. Note that there can be at most p different heavy hitters per attribute. Therefore, there are at most $3p^2$ events to which the set H_4 applies. The overall fraction of events received by each worker is asymptotically equal to the maximum of the three cases, which is $O(1/p^{1/2})$.

5.2 Stream Statistics for Slicing

To adapt the hypercube algorithm to the monitoring setting, we first generalize the notions of relation size and heavy hitters to event streams. Our automatic splitting strategy is selected based on these statistics. Since streams are unbounded, we consider non-overlapping time intervals of a fixed size Δ. Non-overlapping means that all intervals begin at multiples $\theta \cdot \Delta$. We call $\theta \in \mathbb{N}$ the interval's *time index*. The interval size Δ is a parameter of our model.

The choice of Δ represents a tradeoff. Larger values smooth out irregularities in the stream and thus reduce the variability of the characteristics. The downside is lower precision, which can impact monitoring latency. For example, consider a stream where the events are spaced uniformly and can be monitored without additional latency. In the worst-case input with the same event rate, all events in an interval arrive simultaneously, such that one of the events is delayed by the combined processing time of all events. The larger Δ is, the larger is the difference between this maximal latency and the best case.

Recall that an event stream $(\tau_i, D_i)_{i\in\mathbb{N}}$ is an infinite sequence of time-stamped databases. Given an arbitrary event stream and time index θ, the *r-event rate* $\gamma_\theta(r)$ is the average number of events with name $r \in \mathbb{E}$ and a time-stamp in the interval $I_\theta = [\theta \cdot \Delta, (\theta+1)\cdot\Delta)$ per time unit, i.e.,

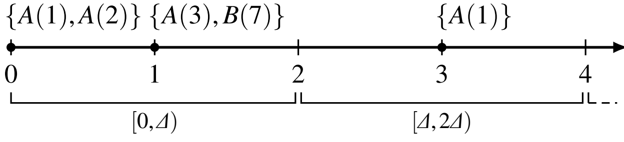
$$\gamma_\theta(r) = \frac{1}{\Delta} \cdot \sum_{\tau_i \in I_\theta} |D_i(r)|.$$

As before, $D_i(r)$ denotes the set of events with the name r in the database D_i. The *event rate* at time θ is $\gamma_\theta = \sum_{r\in\mathbb{E}} \gamma_\theta(r)$, and the *relative r-event rate* is $\gamma'_\theta(r) = \gamma_\theta(r)/\gamma_\theta$. For all names $r \in \mathbb{E}$ and attributes $i \in \{1,\dots,\iota(r)\}$, the *frequency* $F_\theta(d,r,i)$ of $d \in \mathbb{D}$ is

$$F_\theta(d,r,i) = \frac{1}{\Delta} \cdot \sum_{\tau_j \in I_\theta} |\{r(d_1,\dots,d_{\iota(r)}) \in D_j \mid d_i = d\}|.$$

The frequency indicates how often the value d occurs on average in the i-th attribute of r. The set of *heavy hitters* at time θ is $\mathcal{H}_\theta(r,i) = \{d \in \mathbb{D} \mid F_\theta(d,r,i) > \gamma_\theta(r)/p\}$, where $p \in \mathbb{N} - \{0\}$ is a fixed parameter. This follows the definition of heavy hitters for databases from the previous subsection: heavy hitters are those values whose frequency exceeds the threshold $\gamma_\theta(r)/p$. For slicing, we set $p = |K|$, the number of slices.

Example 6 Let $\varDelta = 2$ and $|K| = 2$. Given the prefix



of some event stream, we can infer the following stream statistics: $\gamma_0(A) = \frac{3}{2}$, $\gamma_0(B) = \gamma_1(A) = \gamma_1 = \frac{1}{2}$, $\gamma_1(B) = 0$, $\gamma_0 = 2$, $\mathcal{H}_0(A,1) = \{\}$, $\mathcal{H}_0(B,1) = \{7\}$, and $\mathcal{H}_1(A,1) = \{1\}$.

Let $f \in (V(\varphi) \to \mathbb{D}) \to \mathcal{P}(K)$ be a splitting strategy as in Definition 4. The *load* $\lambda_\theta(k, f)$ of the slice identified by $k \in K$ is the average rate of events in that slice relative to γ_θ, i.e.,

$$\lambda_\theta(k, f) = \frac{1}{\varDelta \cdot \gamma_\theta} \cdot \sum_{\tau_i \in I_\theta} |\{e \in D_i \mid k \in \hat{S}_f(e)\}|.$$

The *maximum load* $\lambda_\theta(f)$ is taken over all slices, $\lambda_\theta(f) = \max_{k \in K} \lambda_\theta(k, f)$.

We consider the problem of finding a splitting strategy that minimizes the maximum load for all event streams with given relative r-event rates, heavy hitters, and number of submonitors. Since these rates and the load are relative to the overall event rate γ_θ, we thus maximize the throughput of the parallelized monitor and the utilization of the submonitors. We do not aim at optimal splitting strategies for arbitrary MFOTL formulas. Instead, we are interested in heuristics providing strategies that are effective in practice. Moreover, we restrict our discussion to event streams with constant relative r-event rates and heavy hitters (constant with respect to θ). Equivalently, the choice of the splitting strategy applies to a single interval of size $\varDelta$. We therefore omit the index θ and write $\gamma(r)$, $\lambda(f)$, and so forth. We have started to address time-varying statistics in a separate work [53].

5.3 Slicing using the Hypercube Algorithm

We instantiate our joint data slicer (Section 4.2) with a strategy that is derived from the hypercube algorithm for database queries (Section 5.1). Observe that monitoring an MFOTL formula without any temporal operators corresponds to evaluating a database query for each index in the event stream. In this case, the subproblem of computing the satisfying valuations at any given index on parallel workers (i.e., submonitors) is solved by the hypercube algorithm. We show below that the mapping phase of the algorithm can be rephrased as a splitting strategy for the joint data slicer. Since we have established this slicer's correctness for all MFOTL formulas, we can thus apply the hypercube approach to temporal formulas and event streams, too.

Recall that several copies of the basic hypercube algorithm are executed in its heavy hitter-aware extension. Each copy sends the event e to a set $T(e)$ of workers (Equation (1) in Section 5.1), which depends on the query q. We will now run all copies in parallel on a single set K of workers. For every variable subset $H \subseteq V(q)$, we assume a bijection $\xi_H : [p_{H,1}] \times \cdots \times [p_{H,n}] \to K$. We can then describe the mapping phase of the extended algorithm by the single equation

$$T'(e) = \bigcup_{v \in matches(q,e)} \{\xi_H(h_{H,1}(v(x_1)), \ldots, h_{H,n}(v(x_n))) \mid H = heavy(q, v)\},$$

such that e is sent to the workers in $T'(e)$. Note that the right-hand side of the equation has the same structure as the one for the joint data slicer $\hat{S}_f(e)$ in Definition 4 once we replace $matches(q,e)$ with $matches(\varphi,e)$. Both these sets contain the valuations for which the event e is potentially relevant, i.e., for which the containment in the query result and the satisfaction of the formula φ, respectively, may depend on e.

To complete the transition from queries to MFOTL formulas, we determine the equivalent of $heavy(q, v)$ for φ. Recall that the set $heavy(q, v)$ contains a variable x if the image of an atom in the query q under v contains a heavy hitter in the corresponding relation. The variable is treated differently because it might not be possible to distribute the relation evenly by hashing the variable. We see that $heavy(q, v)$ depends on the heavy hitters in all events e with $v \in matches(q, e)$. Let $heavy_{var}(\varphi, x)$ be the union of all $\mathcal{H}(r, i)$ for which there is a subformula $r(y_1, \ldots, y_{\iota(r)})$ in φ with $y_i = x$. We then define $heavy(\varphi, v)$ as $\{x \mid v(x) \in heavy_{var}(\varphi, x)\}$.

The following set is nonempty and thus a valid splitting strategy (see Definition 4):

$$f(v) = \{\xi_H(h_{H,1}(v(x_1)), \ldots, h_{H,n}(v(x_n))) \mid H = heavy(\varphi, v)\}.$$

We call f the *hypercube strategy* for φ given $h_{H,j}$, ξ_H, and $\mathcal{H}$.

Algorithm 1 outputs the slice identifiers to which the joint data slicer (Section 4.3) sends a given event according to the hypercube strategy f. We write $\langle\rangle$ for the partial map that is undefined everywhere and codom(h) for h's codomain. Concretely, Algorithm 1 computes the union of $f(v')$ for all v' matching the event. To this end, it computes a partial valuation v for each of the formula's predicates by matching the event with the predicate. The valuation v assigns values to those variables that occur in the predicate. The algorithm subsequently iterates over all full valuations v' (which assign to all free variables) that extend v. This is done in two steps because the set of these valuations may be infinite. First, the algorithm iterates over all $H = heavy(\varphi, v')$, of which there are finitely many. We skip those sets H that contain a variable x with $heavy_{var}(\varphi, x) = \{\}$ because there is no valuation v' where $H = heavy(\varphi, v')$. Second, for each H, the algorithm constructs the finite set of coordinates $(h_{H,1}(v'(x_1)), \ldots, h_{H,n}(v'(x_n)))$ directly by enumerating the codomain of $h_{H,j}$ if x_j is not assigned by v.

What remains is to choose the hash functions $h_{H,j}$ and the mappings ξ_H. As with databases, we select $h_{H,j}$ uniformly at random with a given codomain $[p_{H,j}]$. The *shares* $p_{H,j}$

Input: φ with free variables $x_1, \ldots, x_n$; $(h_{H,i})_{H,i}$, $(\xi_H)_H$, $(heavy_{var}(\varphi, x_i))_i$; event $e = r(d_1, \ldots, d_{\iota(r)})$
Output: slice identifiers $T = \bigcup_{v \in matches(\varphi, e)} f(v)$

1 $T \leftarrow \{\}$;
2 **foreach** *subformula* $r'(y_1, \ldots, y_{\iota(r)})$ *of* φ **do**
3 | **if** $r = r'$ **then** // *compute partial valuation v induced by e*
4 | | $v \leftarrow \langle\rangle$;
5 | | **for** $i \leftarrow 1$ **to** $\iota(r)$ **do**
6 | | | **if** $y_i \in \mathbb{V}$ **then**
7 | | | | **if** $v \neq \bot \land (v(y_i) = \bot \lor v(y_i) = d_i)$ **then** $v \leftarrow v[y_i \mapsto d_i]$ **else** $v \leftarrow \bot$;
8 | | | **else if** $y_i \neq d_i$ **then**
9 | | | | $v \leftarrow \bot$;
10 | | | **end**
11 | | **end**
12 | | **if** $v \neq \bot$ **then** // *recursively enumerate slices for each* $heavy(\varphi, v')$ *where* v' *extends* v
13 | | | $T \leftarrow T \cup$ AllHeavy(v, {}, 1);
14 | | **end**
15 | **end**
16 **end**
17 **Function** AllHeavy(v, H, i) **is**
18 | **if** $i > n$ **then** // *recursively enumerate slices for each* v' *extending* v
19 | | **return** AllExtensions(v, H, $\langle\rangle$, 1)
20 | **else if** $v(x_i) = \bot \land heavy_{var}(\varphi, x_i) \neq \{\}$ **then** // *unknown if* x_i *is a heavy hitter because it is not assigned*
21 | | **return** AllHeavy(v, H, $i+1$) $\cup$ AllHeavy(v, $H \cup \{x_i\}$, $i+1$)
22 | **else if** $v(x_i) \in heavy_{var}(\varphi, x_i)$ **then**
23 | | **return** AllHeavy(v, $H \cup \{x_i\}$, $i+1$)
24 | **else**
25 | | **return** AllHeavy(v, H, $i+1$)
26 | **end**
27 **end**
28 **Function** AllExtensions(v, H, t, i) **is**
29 | **if** $i > n$ **then**
30 | | **return** $\{\xi_H(t)\}$
31 | **else if** $v(x_i) = \bot$ **then** // *x_i not assigned: consider all coordinates*
32 | | **return** $\bigcup_{z \in \mathrm{codom}(h_{H,i})}$ AllExtensions(v, H, $t[i \mapsto z]$, $i+1$)
33 | **else**
34 | | **return** AllExtensions(v, H, $t[i \mapsto h_{H,i}(v(x_i))]$, $i+1$)
35 | **end**
36 **end**

Algorithm 1: Hypercube Strategy

thus parametrize a randomized family of splitting strategies. We select the hash functions anew for every run of the parallel monitor, such that they are independent of the input trace. The mappings ξ_H can be arbitrary; in practice, we map coordinates to slice identifiers in $[p]$:

$$\xi_H(x_1, x_2, \ldots, x_n) = x_1 + p_{H,1} \cdot (x_2 + p_{H,2} \cdot (\cdots (x_{n-1} + p_{H,n-1} \cdot x_n))).$$

Example 7 Assume that there are no heavy hitters in the event stream and that $p = q^2$ for some $q \in \mathbb{N}$. Let $\varphi = P(x_1) \land \CIRCLE\, P(x_2)$ with shares $p_1 = p_2 = q$. We conceptually arrange the slices in a square with side length q. Each P event is assigned to one coordinate in the square's first dimension by the first atom, and to another coordinate in the second dimension by the second atom. Each coordinate is associated with q slices, and there is a single slice that agrees on both coordinates. Therefore, $2q-1$ slices receive the event. The load is approximately $\lambda = (2q-1)/q^2$. The average event rate per slice is lower than the event rate of the input stream if $\lambda < 1$, i.e., $q \geq 2$. This improves over any combination of single variable slicers (see Section 4.3).

Example 8 We extend the triangle query q_T and the database from Example 5 to the formula $\varphi = ((\blacklozenge_{[0,10]} P(x_1, x_2)) \land Q(x_2, x_3)) \land \neg \Diamond_{[0,10]} R(x_3, x_1)$ and some event stream with $\gamma(P) = \gamma(Q) = \gamma(R) = m$, having $\mathcal{H}(P, 1) = \{0\}$ as the only heavy hitter. We can reuse the optimal shares from Example 5 because q_T and φ consist of the same atoms, and the stream statistics correspond to the database statistics. Let $p = 64$. We simplify the hash functions to the modulus (e.g., $h_{\{x_1\},2}(x) = x \bmod 8$, since $p_{\{x_1\},2} = p^{1/2} = 8$). Before applying the mappings ξ_H, we obtain the following assignment of events to coordinate vectors $(h_{H,1}(v(x_1)), h_{H,2}(v(x_2)), h_{H,3}(v(x_3)))$. Note that there are no coordinates for all other H, since $heavy_{var}(\varphi, x)$ is nonempty only for $x = x_1$.

event	coordinates for $H = \{\}$	coordinates for $H = \{x_1\}$
$P(0,1)$	—	010, 011, 012, 013, 014, 015, 016, 017
$P(1,1)$	110, 111, 112, 113	—
$Q(1,7)$	013, 113, 213, 313	017
$R(7,0)$	—	007, 017, 027, 037, 047, 057, 067, 077

If these events are within 10 time units of each other, the valuation $\langle x_1 = 0, x_2 = 1, x_3 = 7\rangle$ will be recognized successfully as satisfying: the events $P(0,1)$, $Q(1,7)$, and $R(7,0)$ are all part of the slice with the identifier $\xi_{\{x_1\}}(017) = 57$.

We apply an additional optimization to the hash functions. The shares for two variable subsets $H_1 \neq H_2$ may be equal and hence there is no need to distinguish them. This occurs if the variables in the symmetric difference of H_1 and H_2 receive a share of 1. If we choose the hash functions independently, however, there is a large probability that the slice sets computed with H_1 and H_2 differ for a given event. We reduce this unnecessary event duplication by using the same hash functions for H_1 and H_2, as shown in Example 9 below.

Example 9 Let $\varphi = P(x_1) \land Q(x_1, x_2)$, $p_{\{\},1} = p_{\{x_2\},1} = 2$, and $p_{H,2} = 1$ for all H. Assume that the attribute x_1 of either event has no heavy hitters. If $h_{\{\},1}$ and $h_{\{x_2\},1}$ are independent hash functions, the events are duplicated with probability 1/2. If $h_{\{\},1} = h_{\{x_2\},1}$, each event is sent to only one of the two slices, which reduces the expected maximum load by a third.

We can transfer the load analysis by Beame et al. [21, Theorem 3.2] and Koutris et al. [39, Theorem 2] from the database setting to ours. This allows us to use the algorithm of

Chu et al. [27] to optimize the shares. The transfer is based on the following observation: applying our hypercube strategy algorithm to an interval of an event stream incurs the same load as using the hypercube algorithm (Section 5.1) on the database constructed from that interval. This database is the (multiset) union of all databases in the stream that belong to the interval.[2] Therefore, relation sizes correspond to r-event rates $\gamma(r)$. We overapproximate the load by summarizing the partial loads induced for each choice of the variable set H. We further simplify the analysis by using the rate $\gamma(r)$ for each H, even though only a subset of the r-events may be sliced according to this H. Let $r(y_1,\ldots,y_{\iota(r)}) \preceq \varphi$ denote the fact that $r(y_1,\ldots,y_{\iota(r)})$ is a subformula of φ. The maximum load λ is bounded from above by

$$\hat{\lambda} = \frac{1}{\gamma} \cdot \sum_{\substack{H\subseteq V(\varphi),\\ r(y_1,\ldots,y_{\iota(r)})\preceq\varphi}} \frac{\gamma(r)}{\prod_{x_i\in\{y_1,\ldots,y_{\iota(r)}\}\cap\mathbb{V}} p_{H,i}} = \sum_{\substack{H\subseteq V(\varphi),\\ r(y_1,\ldots,y_{\iota(r)})\preceq\varphi}} \frac{\gamma'(r)}{\prod_{x_i\in\{y_1,\ldots,y_{\iota(r)}\}\cap\mathbb{V}} p_{H,i}}$$

with high probability over the random choice of the hash function, up to a factor logarithmic in p. (We divide by γ because we have defined the load relative to γ.)

Algorithm 2 optimizes the shares and selects the hash functions. For each $H \subseteq V(\varphi)$, it first iterates over all valid share vectors $p_H = (p_{H,1},\ldots,p_{H,n})$, where a share vector is called valid if $\prod_{1\le i\le n} p_{H,i} \le p$, and $p_{H,i} = 1$ for all $x_i \in H$. Note that we allow the shares' product to be smaller than p, which may be beneficial if p cannot be factorized optimally [27]. The maximal number of submonitors p is the input to the optimization, together with the relative r-event rates $\gamma'(r)$. We choose the share vector with the smallest value for

$$\mathrm{Cost}(p_H) = \sum_{r(y_1,\ldots,y_{\iota(r)})\in\varphi} \frac{\gamma'(r)}{\prod_{x_i\in\{y_1,\ldots,y_{\iota(r)}\}\cap\mathbb{V}} p_{H,i}},$$

thereby minimizing $\hat{\lambda}$. We adopt a heuristic by Chu et al. [27] and break ties by choosing the vector with smallest maximum share $\max_i p_{H,i}$. This favors a more even distribution of shares to increase resilience against heavy hitters that are not accounted for in the statistics provided. Once the shares have been computed, Algorithm 2 samples random hash functions RandomHash(q) with codomain $[q]$. It implements the optimization mentioned above, where the hash functions with the same codomain are reused.

```
Input: φ with free variables x_1,...,x_n; number of submonitors
       p, relative event rates (γ'(r))_r
Output: parameters (h_{H,i})_{H,i} for the hypercube strategy
 1 foreach H ⊆ V(φ) do                        // search for best shares
 2 |   p_H ← (1,...,1);
 3 |   OptimizeShares(H, 1, p, (1,...,1));
 4 end
 5 foreach q ∈ {p_H | H ⊆ V(φ)} do            // share hash functions
     among equal variable subsets
 6 |   for i ← 1 to n do
 7 |   |   h ← RandomHash(q_i);
 8 |   |   foreach H ⊆ V(φ), p_H = q do h_{H,i} ← h;
 9 |   end
10 end
11 Procedure OptimizeShares(H, i, p, c) is
12 |   if i ≤ n then
13 |   |   if x_i ∈ H then
14 |   |   |   OptimizeShares(H, i+1, p, c[i ↦ 1]);  // p_{H,i} is
   |   |   |     always 1 if x_i ∈ H
15 |   |   else
16 |   |   |   for c ← 1 to p do OptimizeShares(H, i+1,
   |   |   |     ⌊p/c⌋, c[i ↦ c]);
17 |   |   end
18 |   else
19 |   |   if Cost(c) < Cost(p_H) ∨ (Cost(c) =
   |   |     Cost(p_H) ∧ max_i c_i < max_i p_{H,i}) then p_H ← c;
20 |   end
21 end
```

Algorithm 2: Hypercube Optimization

5.4 Discussion

Algorithm 1, which computes the hypercube strategy, iterates over all combinations of the formula's predicates with the subsets of its free variables. For each combination, it enumerates up to p slice identifiers. Therefore, Algorithm 1's complexity is bounded by $O(|\varphi| \cdot 2^n \cdot n \cdot p)$, where $|\varphi|$ is the size of the formula φ and n is the number of free variables in φ. We assume $n, p \ge 1$ and that all operations that involve $\mathbb{D}$ and slice identifiers in $[p]$ are computed in $O(1)$ time, including the hash functions. The linear factor p is unavoidable: events may need to be broadcast to all p slices, e.g., if their arity is zero. The exponential complexity in n stems from the generic treatment of heavy hitters.

A possible optimization is to enumerate only subsets of those variables x_i which have a share $p_{H,i} > 1$ for some H. This does not decrease the complexity for all formulas though. By bounding the number of possible share combinations with product q from above by $n^{\log_2 q}$, we find that Algorithm 2's complexity is in $O(|\varphi| \cdot (4^n \cdot n + 2^n \cdot p \cdot n^{\log_2 p}))$. The 4^n factor can be improved to 2^n by avoiding the innermost loop in line 8 and by iterating over the list of p_H in lexicographic order instead (lines 5–10). We omit this optimization for clarity. Note that Algorithm 2 runs only once when the monitor is initialized, whereas Algorithm 1 is invoked for every event.

The minimum possible load achieved using the hypercube strategy depends on the pattern of free variables in the formula's atoms. A detailed discussion is provided by Koutris et al. [39]. The ideal case is a formula in which all atoms with a significant event count share a variable, together with a stream that never assigns a heavy hitter to that variable.

[2] We use multisets because events may be repeated at different indices. The upper bound from Beame et al. [21] extends to multisets.

Then the load per slice is $1/p$. Atoms with missing variables, and equivalently variables with heavy hitters, increase the fraction to $1/p^z$ for some exponent $z > 1$.

The (worst-case) optimality of the hypercube algorithm for conjunctive queries does not extend to full MFOTL. This already becomes evident for simple non-temporal formulas with disjunctions, such as $P(x_1,x_2) \wedge (Q(x_1) \vee Q(x_2))$. If $\gamma(P) = \gamma(Q)$ and in the absence of heavy hitters, our approach will have load $(\sqrt{p} + \frac{1}{2})/p \approx 1/\sqrt{p}$ with p submonitors. However, the formula is equivalent to $(P(x_1,x_2) \wedge Q(x_1)) \vee (P(x_1,x_2) \wedge Q(x_2))$, and thus we can process each disjunct independently. By using the optimal hypercube strategy for each disjunct (with shares $p_1 = p$ and $p_2 = p$, respectively), we would obtain a total load of $2/p$, which is asymptotically better. The load can be further improved to $3/(2p)$ by using the same hash function for x_1 in the first and x_2 in the second disjunct, such that the Q events are not duplicated.

Overall, it is unclear how this technique can be generalized to MFOTL formulas with arbitrarily nested temporal operators. In general, optimality for arbitrary formulas is out of reach because it would require us to decide MFOTL: if the formula is contradictory, the best possible slicer simply drops all events. We therefore settle for a more pragmatic solution and only focus on syntactic aspects of the formulas' structure.

We assumed that the submonitors' throughput does not depend on the events. It was therefore sufficient to minimize the load to optimize the throughput. This simplification is not always appropriate for monitors like MonPoly. The reason is that MonPoly constructs intermediate results, whose size depends on the monitor's input and which affects the complexity of further operations inside the monitor. It might be possible to achieve even higher throughput by taking the events' distribution and its impact on the monitoring performance into account. We leave such optimizations for future work.

In contrast to offline monitoring, *stream* statistics (such as $\gamma(r)$ and $\mathcal{H}(r,i)$) cannot be obtained for the entire stream. Still, our approach assumes that these statistics are already available before the start of monitoring. In practice, this is a reasonable assumption since organizations have access to historical data that can serve as a good source of representative stream statistics before staring online monitoring.

Moreover, the statistics may change over time. In this case, one must obtain stream statistics during monitoring. This can be done using approximate algorithms [28], which have minimal impact on monitoring's performance. Furthermore a reasonable extension of the slicing framework is to adaptively modify the splitting strategy whenever the statistics change significantly. Thus, the monitor could start with a default strategy and refine it as more data is processed. (Event-separable slicers as defined in Section 4.2 cannot be adaptive because they must behave uniformly on the event stream.) We have already made first steps towards computing stream statistics online [31] and performing adaptive slicing [53].

Our approach affects only the event rate, but not the index rate, which is the number of databases per unit of time. The index rate impacts the performance of monitors such as MonPoly because each database triggers an update step. For a syntactic fragment of MFOTL, MonPoly reduces the number of update steps skipping empty databases [11]. In this case, we could already filter empty databases in the splitter.

6 Implementation

We implemented a parallel online monitoring framework based on the joint data slicer and built on top of the Apache Flink stream processing framework. The source code consists of roughly 3,100 lines of Java and Scala and is publicly available [54]. Given a formula, our framework instantiates a parallel online monitor, which then reads events from a TCP socket or a text file, monitors the events in parallel, and writes all satisfying valuations to an output socket or file. The parallel monitor delegates the monitoring of individual slices to external tools, called *submonitors*. Our implementation supports the tools MonPoly [16] and DejaVu [37] as submonitors.

To instantiate a parallel online monitor, our framework uses the Flink API to construct a dataflow graph, whose nodes are stream operators. These operators retrieve data streams from external sources, apply processing functions to stream elements, and output the elements to sinks. Operators can execute in parallel. Stream elements can be partitioned according to user-specified keys. At runtime, Flink deploys the graph to a distributed computing cluster. We chose Flink for its low latency stream processing and its support for fault-tolerant computing. Fault tolerance is ensured using a distributed checkpointing mechanism [25]. The system recovers from failures by restarting from regularly created checkpoints. Operators must therefore expose their state to the framework to enable checkpointing.

The inputs to our monitoring framework are the formula, the number and type of parallel submonitors, the stream statistics for the shares' optimization, and the heavy hitter values. The framework precomputes the shares using Algorithm 2 and creates a parallel monitor instance as the dataflow graph shown in Figure 4, where each node is labeled with a Flink operator (e.g., `flatMap`) and a description of its functionality.

During the dataflow's execution, the input events are read, line by line, as strings. We support both MonPoly's and DejaVu's input formats, as well as the CSV format used in the RV competition [8]. The parser then converts the input lines into an internal datatype that stores the event name and the list of data values. The parser's results are flattened into a stream of single events because a single line in MonPoly's format may describe several events at once.

After parsing, the splitter computes the set of target slices for each event. To do so, it executes Algorithm 1 using the optimized shares, precomputed by the framework, and heavy

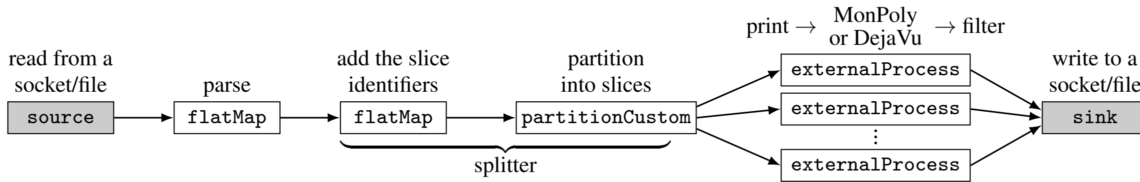


Fig. 4 Our parallel online monitor's dataflow graph

hitter sets as well as the heavy hitter values. For each event and each of its target slices, a copy of the event is sent to the next operator along with the target slice identifier. Then, the stream is partitioned into slices based on the slice identifiers and the slices are sent to the parallel submonitors.

We use the custom `externalProcess` operator in each parallel flow. This operator is responsible for initiating and interacting with an external process, in our case MonPoly or DejaVu. The operator prints, in MonPoly or DejaVu format, one database at a time to the standard input of the external process. (For DejaVu, which expects exactly one event at a time, empty databases are encoded as an event with a name that does not occur in the formula.) The operator simultaneously reads verdicts from the standard output of the process and applies the intersection from J_f's definition (Definition 4), thereby filtering the monitor's output. Finally, all remaining verdicts are combined into a single stream, which is written to an output socket or file.

The above communication with the external process is asynchronous with respect to the Flink pipeline, which prevents these operations from blocking other operators. Flink's `AsyncWaitOperator` supports asynchronous requests to external processes, but it does not manage their state. To optionally provide fault-tolerance, we must checkpoint the submonitors' states because they summarize the events seen so far. Our implementation of the `externalProcess` operator extends the `AsyncWaitOperator` with an interface to retrieve and restore an external state.

We have extended MonPoly with control commands that implement the interface for retrieving and restoring an external state. Whenever Flink instructs the `externalProcess` operator to create a checkpoint, the operator first waits until all prior events have been processed. Then, the command for saving the state is sent to the external process. In response, MonPoly writes its state to a temporary file. The part of the monitor's output received after the checkpoint instruction's arrival at the `externalProcess` operator is also included in the checkpoint. This ensures that no output is lost when other operators create their own checkpoint concurrently. We did not implement a state interface for DejaVu, since we opted to use DejaVu in a black-box manner to demonstrate our framework's generality. Therefore, our parallel monitor is currently not fault-tolerant if DejaVu is used as a submonitor. We conjecture that implementing the state interface in DejaVu is possible with modest effort.

DejaVu monitors closed formulas only and reports violating instead of satisfying valuations. Therefore, when using DejaVu, our framework first closes the input formula φ by adding a prefix of existential quantifiers. Then it negates the closed formula before passing it to the parallel monitor. Thus it ensures that DejaVu's output is consistent with MonPoly's output whenever they are used as submonitors within our framework. The splitter uses the original formula φ because it is only effective if there are free variables. As the output of DejaVu consists only of the violating indices for the closed and negated formula, we cannot compute the intersection from J_f's definition with φ's valuations. Hence, we must use the simplified joiner J', which is correct under the assumptions of Theorem 3. This limits the applicability of our approach using DejaVu to monitor certain formulas, and we cannot account for heavy hitters because otherwise the hypercube strategy would not satisfy condition *(4)* of Lemma 3.

The parts of the dataflow preceding the submonitors currently operate sequentially. This is a bottleneck that limits scalability, since all input events must be processed sequentially by the splitter. Despite this limitation of our implementation, the splitter and the surrounding operators could be parallelized too: Our splitter processes events separately because it implements the event-separable joint data slicer (Section 4.2). A parallel splitter would be particularly effective if the event source itself is distributed. However, we must ensure that events arrive at the submonitors in chronological order. This order is no longer guaranteed if the splitter is partitioned into concurrent tasks. In a separate line of work [13], we propose a possible solution that buffers and reorders events before forwarding them to each submonitor.

7 Evaluation

We structure our evaluation to answer the following research questions, which assess the scalability, practicality, overhead, and generality of our framework.

$\mathrm{RQ}_{\mathit{frm}}$: How does our monitor scale for different formulas?

$\mathrm{RQ}_{\mathit{rate}}$: How does our monitor scale with respect to the index rate and the event rate?

$\mathrm{RQ}_{\mathit{stats}}$: Can knowledge about the relative event rates improve performance?

$\mathrm{RQ}_{\mathit{skew}}$: Can knowledge about heavy hitter values improve performance?

RQ_{real}: Are the scalability improvements applicable to real-world monitoring use cases?

RQ_{oh}: How much overhead is incurred by using our framework? Specifically, how does it compare to the standalone tools MonPoly and DejaVu?

RQ_{ft}: How much overhead is incurred by supporting fault tolerance (FT)?

RQ_{gen}: Can our framework scale with different submonitors?

The scalability (RQ_{frm} and RQ_{rate}) of our framework is its ability to handle growing event rates by using more submonitors. This includes the framework's ability to leverage its knowledge about the event stream to further improve monitoring performance (RQ_{stats} and RQ_{skew}). The framework is practical (RQ_{real}) if it can be used in a real-world setting, i.e., to scalably monitor a real event stream. The overhead of the framework is the fraction of its time and memory usage that is not spent on running the submonitors (RQ_{oh} and RQ_{ft}). Finally, the framework's generality is its ability to be used with different first-order (sub)monitors (RQ_{gen}).

To answer the above questions, we organize our evaluation into two families of experiments, each monitoring a different type of input stream, either synthetic or real-world. The synthetic streams are used to analyze the effects of individual parameters, such as the event rate, whereas the real-world streams attest to our framework's ability to scalably solve realistic problems. Figure 5 summarizes the parameters used for each experiment, which we explain next.

Synthetic Experiments. In the experiments with synthetic streams (Figure 5), we monitor the three formulas *star*, *linear*, and *triangle* and their past-only, non-metric variants *star-past*, *linear-past*, and *triangle-past* (Figure 6). Different occurrence patterns of free variables in the formulas are used to test RQ_{frm}. The formulas cover common patterns in database queries [21], which we additionally extend with temporal operators. We focus on variable occurrence patterns over other formula features (e.g., formula size) since they affect our framework directly, rather than just the submonitors.

We have implemented a stream generator tailored to each of the three formulas. The generator takes a random seed and synthesizes streams with configurable characteristics. Specifically, the synthesized streams on average have constant characteristics across all time indices θ. The streams contain binary events labeled with P, Q, or R and have configurable event rates and index rates. This setup allows us to test RQ_{rate}.

Figure 5 summarizes the event rates used in our experiments. Note that we evaluate only those combinations of event rates and number of submonitors that do not take too long to execute. Specifically, we limit individual monitoring runs to 5 minutes of total execution time. For example, in the *Synthetic*^{MonPoly} experiments, we monitor the *star* formula with the standalone MonPoly instance on streams with event rates up to 20 000 (denoted as 20k in Figure 5).

To test RQ_{stats} and RQ_{skew}, the generator can also synthesize streams with configurable relative event rates ($\gamma'_\theta(P)$, $\gamma'_\theta(Q)$, $\gamma'_\theta(R)$) and force some event attribute values to be heavy hitters. Attribute values are sampled from two possible types of distributions. Non-heavy hitter values are selected uniformly at random from the set $\{0, 1, \ldots, 10^9 - 1\}$; heavy hitter values are drawn from a Zipf distribution. The Zipf distribution's probability mass function is $p(x) = x^{-z} / \sum_{n=1}^{10^9} n^{-z}$ for $x \in \{1, 2, \ldots, 10^9\}$, i.e., the larger the exponent $z > 0$ is, the fewer values have a large relative frequency. To prevent excessive monitor output, all Zipf-distributed values of R events are increased by 10^6. The distribution type (uniform or Zipf) and the exponent z are defined per variable x (the exponent is thus denoted z_x) and can be supplied as inputs to the generator.

All synthetic streams in our experiments are generated with relative event rates $\gamma'_\theta(P) = 0.01$ and $\gamma'_\theta(Q) = \gamma'_\theta(R) = 0.495$ and with attribute values sampled uniformly at random. In the *Synthetic*$^{heavy\ hitters}$ experiments (Figure 5), we also generate streams with heavy hitter values in valuations of variable a in the *star* formula and variable b in the *linear* and *triangle* formulas, with their Zipf exponents set to 2.

Real-world Experiments. To test RQ_{real}, we use logs from Nokia's Data Collection Campaign [14]. The campaign collected data from the mobile phones of 180 participants and propagated the data between three databases, `db1`, `db2`, and `db3`. The phones uploaded the data directly to `db1`, then a synchronization script `script1` periodically copied the data from `db1` to `db2`. Next, `db2`'s triggers anonymized and copied the data to `db3`. The participants could query and delete their own data from `db1`. Deletions were propagated to all databases.

To obtain streams suitable for online monitoring, we have developed a tool (called *replayer*) that replays log events and simulates the event rate at the log creation time, which is captured by the events' time-stamps. The tool can also replay the log proportionally faster than its event rate, which is useful for evaluating the monitor's performance while retaining the log's other characteristics. Since the log from the campaign spans a year, to evaluate our tool in a reasonable amount of time, we pick a one day fragment with a high average event rate from the log, starting at time-stamp 1 282 921 200. We use the replayer to accelerate the fragment up to 5 000 times. The fragment contains roughly 9.5 million events with an average event rate of 110 events per second. Using the acceleration, we have subjected our tool to streams of over half a million events per second. The logs used [1] and the scripts that synthesize and replay streams [54] are publicly available.

We monitor the formulas *insert*, *delete*, and *custom* (Figure 6). The formulas *insert* and *delete* come from Nokia's Data Collection Campaign, where they proved to be challenging to monitor. Specifically, the two formulas are the negated versions of the *ins-1-2* and *del-1-2* formulas from Basin et al.'s formalization [14], which require a large amount of

Experiment group	Synthetic streams			Real-world streams	
	Synthetic^{MonPoly}	*Synthetic*^{DejaVu}	*Synthetic*$^{heavy\ hitters}$	*Nokia*^{MonPoly}	*Nokia*^{DejaVu}
Tools	MonPoly	MonPoly, DejaVu	MonPoly	MonPoly	MonPoly, DejaVu
Formulas	*star*, *linear*, *triangle*	*star-past*, *linear-past*, *triangle-past*	*star*, *linear*, *triangle*	*insert*, *delete*, *custom*	*custom*
Submonitors	1, 4, 8, 16	1, 4, 8, 16	4, 8, 16	1, 2, 4, 8	1, 2, 4, 8
Event rates (1/s)	10k, 15k, 20k, 25k, 30k, 35k, 40k, 45k, 50k, 55k, 60k, 65k, 70k, 75k	1k, 2k, 4k, 6k, 10k, 15k, 20k, 30k, 50k, 60k	50k	a one day fragment from the Nokia log	a one day (linearized) fragment from the Nokia log
Index rates (1/s)	1, 1000	equal to event rate	1		
Relative event rates	$\gamma'_\theta(P) = 0.01$, $\gamma'_\theta(Q) = 0.495$, $\gamma'_\theta(R) = 0.495$	$\gamma'_\theta(P) = 0.01$, $\gamma'_\theta(Q) = 0.495$, $\gamma'_\theta(R) = 0.495$	$\gamma'_\theta(P) = 0.01$, $\gamma'_\theta(Q) = 0.495$, $\gamma'_\theta(R) = 0.495$		
Value distributions	uniform	uniform	uniform, Zipf with $z_a = 2$ for *star*, Zipf with $z_b = 2$ for *linear* and *triangle*		
Time span	60 s	60 s	60 s		
Total events	event rate × 60 s	event rate × 60 s	event rate × 60 s	9.5 million	9.5 million
Accelerations	1	1	1	1k, 2k, 3k, 4k, 5k	500, 1k, 1.5k, 2k
Stages	online, offline	online, offline	offline	online	online
Fault tolerance	yes, no	no	no	yes, no	no

Fig. 5 Summary of parameters used in our experiments

$$
\begin{aligned}
star &= \big((\blacklozenge_{[0,10s)} P(a,b)) \land Q(a,c)\big) \land \Diamond_{[0,10s)} R(a,d) & star\text{-}past &= \big(\blacklozenge_{[0,\infty)} ((\blacklozenge_{[0,\infty)} P(a,b)) \land Q(a,c))\big) \land R(a,d)\\
linear &= \big((\blacklozenge_{[0,10s)} P(a,b)) \land Q(b,c)\big) \land \Diamond_{[0,10s)} R(c,d) & linear\text{-}past &= \big(\blacklozenge_{[0,\infty)} ((\blacklozenge_{[0,\infty)} P(a,b)) \land Q(b,c))\big) \land R(c,d)\\
triangle &= \big((\blacklozenge_{[0,10s)} P(a,b)) \land Q(b,c)\big) \land \Diamond_{[0,10s)} R(c,a) & triangle\text{-}past &= \big(\blacklozenge_{[0,\infty)} ((\blacklozenge_{[0,\infty)} P(a,b)) \land Q(b,c))\big) \land R(c,a)
\end{aligned}
$$

$$
\begin{aligned}
insert &= (insert(u,\mathsf{db1},pid,dt) \land dt \not\approx \mathsf{unknown}) \land \big(\neg \Diamond_{[0,30h)} \exists u'.\, (insert(u',\mathsf{db2},pid,dt) \lor delete(u',\mathsf{db1},pid,dt))\big)\\
delete &= \Big(\big((delete(u,\mathsf{db1},pid,dt) \land dt \not\approx \mathsf{unknown}) \land (\neg \blacklozenge_{[0,30h)} \exists u',p'.\, insert(u',\mathsf{db1},p',dt))\big) \lor\\
&\quad \big((delete(u,\mathsf{db1},pid,dt) \land dt \not\approx \mathsf{unknown}) \land ((\blacklozenge_{[0,30h)} \exists u',p'.\, insert(u',\mathsf{db2},p',dt)) \lor \Diamond_{[0,30h)} \exists u',p'.\, insert(u',\mathsf{db2},p',dt)))\big)\Big) \land\\
&\quad (\neg \Diamond_{[0,30h)} \exists u',p'.\, delete(u',\mathsf{db2},p',dt))\\
custom &= \exists u_1\ db_1.\ select(u_1,db_1,pid_1,dt) \land \big(\blacklozenge_{[0,\infty)} \exists u_2\ db_2.\ insert(u_2,db_2,pid_2,dt)\big)
\end{aligned}
$$

Fig. 6 MFOTL formulas used in the evaluation

memory when monitored by a single MonPoly instance. We used our knowledge of the data set also to craft the past-only, non-metric *custom* formula with an expensive temporal join involving the (very frequently occurring) *insert* event.

Since we monitor only a one day fragment of the Nokia log, we must initialize our monitor with the appropriate state to obtain the correct output. Therefore, we monitor each formula once on the part of the log preceding the chosen fragment and spanning an appropriate amount of time as defined by each formula's temporal reach. We store the monitor's state obtained at the end of the proceeding fragment and initialize the monitor with the stored state in the experiments.

We have additionally computed the relative event rates for all events, and identified all heavy hitter values in the one day fragment of the Nokia log. We run our framework both with and without this information to answer RQ_{stats} and RQ_{skew}.

Monitors. To test RQ_{oh} and RQ_{gen}, we use MonPoly and DejaVu as parallel submonitors within our framework, and also as standalone monitors for comparison. To accommodate DejaVu, which implements a slightly different monitor function than MonPoly, we need to adapt the parameters of our two families of experiments (see the *Synthetic*^{DejaVu} and *Nokia*^{DejaVu} experiments in Figure 5). First, we use the formulas *star-past*, *linear-past*, *triangle-past*, and *custom* (Figure 6), which belong to the past-only non-metric fragment of MFOTL supported by DejaVu. The formulas are closed and negated prior to invoking DejaVu, since it only monitors closed formulas and just reports violations. DejaVu expects input streams without time-stamps and with databases containing exactly one event. Thus, we modify the streams in our experiments accordingly: each database with more than one event is *linearized*, i.e., translated into a sequence of singleton databases with all time-stamps set to 0. The verdicts of the used formulas are not affected by this transformation.

Moreover, we run the experiments both with and without Flink's fault tolerance mechanism to determine its impact on performance (RQ_{ft}). This is only done when MonPoly is the submonitor, since DejaVu does not support checkpointing.

Measurements. We ran all our experiments on a server with two sockets, each containing twelve Intel Xeon 2.20GHz CPU cores with hyperthreading, which effectively gives us 48 independent computation threads.

To assess our framework's scalability, we measure the (maximal) latency and throughput achieved during our experiments. Latency is the difference between the time a monitor consumes an event and the time it is done processing it. Throughput is the number of events that a monitor processes in a unit of time. We use the wall-clock time values provided by the UNIX `time` command to measure the total execution time, i.e., the time between the moment when the replayer starts emitting events to the monitor and the moment the monitor processes the last emitted event. We also measure the execution time and maximal memory usage of each submonitor. To measure the latency during execution, our replayer injects

a special event, called a *latency marker*, into the stream. Every second, the replayer generates a latency marker, which is tagged with the current time. The marker is then propagated by our framework, preserving its order with respect to the databases containing other events from the input stream. We measure the latency at the framework's output by comparing the current time with the time in the marker's tag. Besides measuring the current latency, we also calculate the maximum latency up to the current point in the experiment.

Since MonPoly's unit of input is a database of events (rather than a single event), it does not perform any processing before it receives an entire database. Its particular input format allows MonPoly to detect that the currently received database is complete only once the first event from the next database is received. This means that our latency measurements as described above would treat the timestamp difference between two consecutive databases in the input as the monitor's processing latency. Thus, we task our replayer to additionally send *watermark* events as part of the input, signaling to MonPoly whenever the currently received database is complete. This effectively allows us to measure the monitor's exact processing time latency, excluding any delay introduced by the delays already present in the input.

When the latency is higher than one second, the latency marker gets delayed too and a timely value cannot be produced. Flink reports zeros for the current latency in this case, while we consider the latest non-zero value. This significantly reduces the noise in our measurements.

In addition to *online* experiments, where we use our replayer to simulate event streams, we also execute all our synthetic experiments *offline*. Specifically, we directly supply the monitored log as a file to the monitor. The monitor consumes the log at a rate defined by its current processing speed. We can then calculate our framework's throughput as the ratio of the total number of events and the measured offline execution time. The stage [30] (offline or online) at which we run our monitor in each of the experiments is specified in Figure 5.

Since we focus on performance measurements, we discard the tool's output during all of our experiments. Each run of a monitor with a specific configuration is repeated three times and the collected metrics are averaged to minimize the noise in the measurements.

Results. Figure 7 shows the results of using our framework with MonPoly to monitor synthetic streams. We show the results when fault tolerance is enabled, since they are less favorable for our framework. Plots labeled with Tool^N denote that our framework used N instances of Tool as submonitors. Omitting the number of submonitors indicates a standalone run of the Tool. Our experiments demonstrate our framework's low overhead (RQ_{oh}): a standalone run of a Tool exhibits the same performance as a run of our framework with one submonitor (Tool^1).

Figure 7a shows the achieved throughput (top), the maximum latency (middle), and the maximal memory consumption across all submonitors (bottom) when monitoring the formula *star* with different numbers of submonitors. For example, our tool exhibits a latency of 27 seconds for an event rate of 15 000 events per second if a single submonitor is used. Similar latency is exhibited with 4 submonitors when monitoring events rates above 45 000 events per second. In contrast, using 16 submonitors achieves sub-second latency for all event rates in our experiments. With an increasing number of submonitors, each submonitor receives fewer events and hence uses less memory, while collectively the submonitors handle larger throughput. This experiment answers RQ_{rate}: our tool handles significantly higher event rates by using more parallel submonitors.

Figure 7b shows the achieved throughput (top), the maximum latency (middle), and the maximal memory consumption (bottom) of our tool when monitoring *star*, *triangle*, and *linear* formulas using 4 submonitors. The plots show six graphs, where each graph shows the results of monitoring one of the three formulas over a stream with an index rate of either 1 or 1 000. Since the index rate affects the performance of MonPoly [15], the overall framework is also affected (RQ_{rate}). The event rate gain due to parallel monitoring depends on the variable occurrence patterns in the monitored formula (RQ_{frm}). Namely, the variable pattern in the *star* formula is the one that exhibits the best scalability due to variable a's occurrence in all the formula's atoms.

In the experiments described so far, we did not supply our framework with the relative event rates for the event names in the stream. Figure 7c positively answers RQ_{stats} by showing that our tool's performance substantially increases when using 4 and 8 submonitors and when the statistics about the stream are known in advance. We use Tool^N_{stats} to denote that our framework runs Tool on N submonitors with relative event rates provided ahead of time.

Figure 8 shows the results of the same experiments as in Figure 7 but now using our framework with DejaVu as the submonitor. Fault tolerance was disabled in these experiments. Similarly as before, the experiments show that our framework can handle higher event rates by using more parallel submonitors (RQ_{rate}). Regarding RQ_{gen}, our results demonstrate improved throughput, latency, and memory consumption with two different first-order monitors. Both Figure 7a and Figure 8a answer RQ_{oh}: they show that our framework achieves better performance than MonPoly and DejaVu on their own, except when only a single submonitor is used, where it exhibits essentially the same performance.

Figure 9 summarizes the results of using our framework with MonPoly to monitor the real-world log from the Nokia case study (RQ_{real}). The event and index rates are defined by the log; we only control the acceleration used by the replayer. As we anticipated earlier, the *custom* formula is the hardest

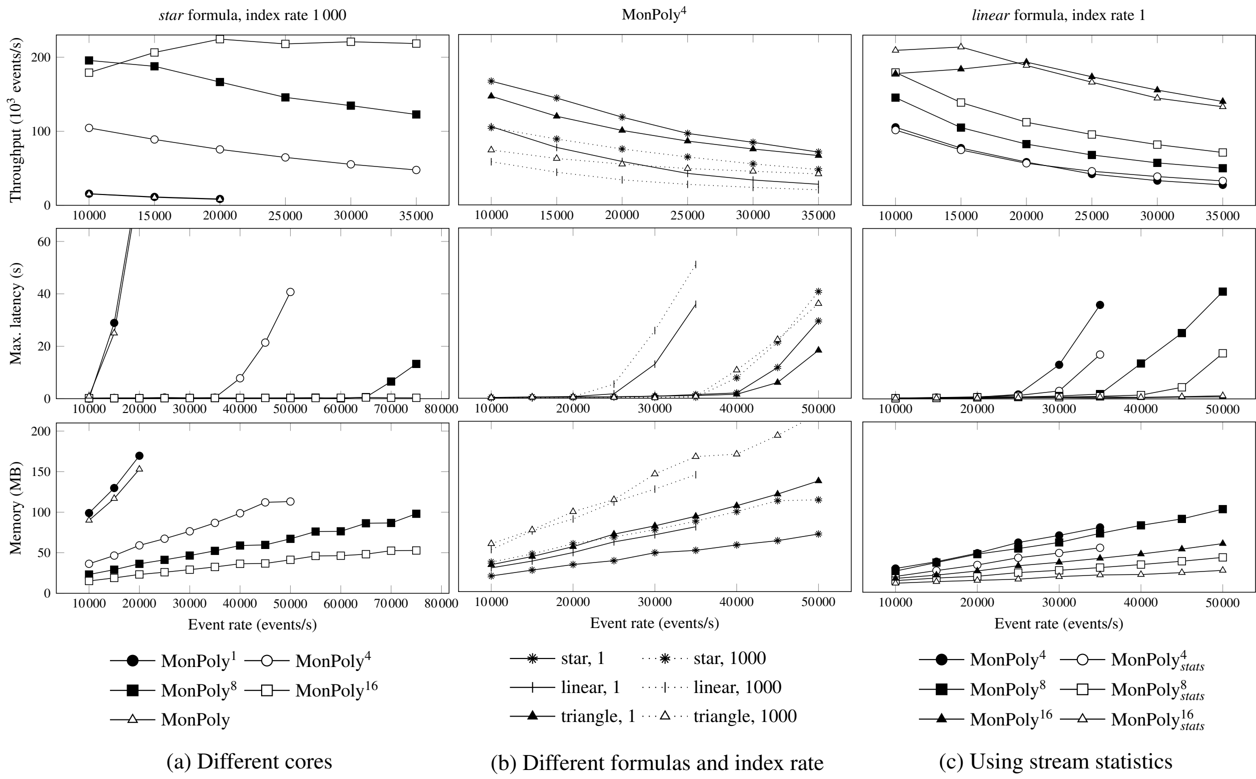


Fig. 7 *Synthetic*^{MonPoly} experiments: monitoring synthetic streams with MonPoly and with fault tolerance

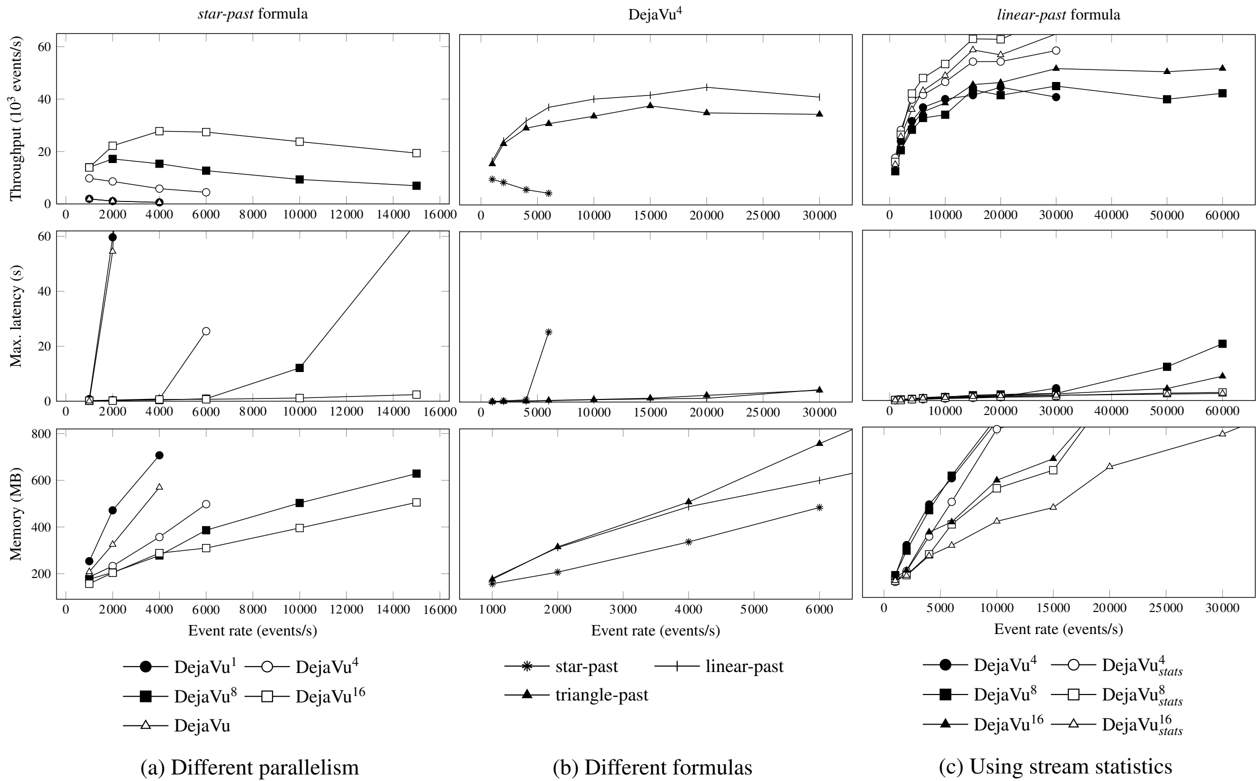


Fig. 8 *Synthetic*^{DejaVu} experiments: monitoring synthetic streams with DejaVu and without fault tolerance

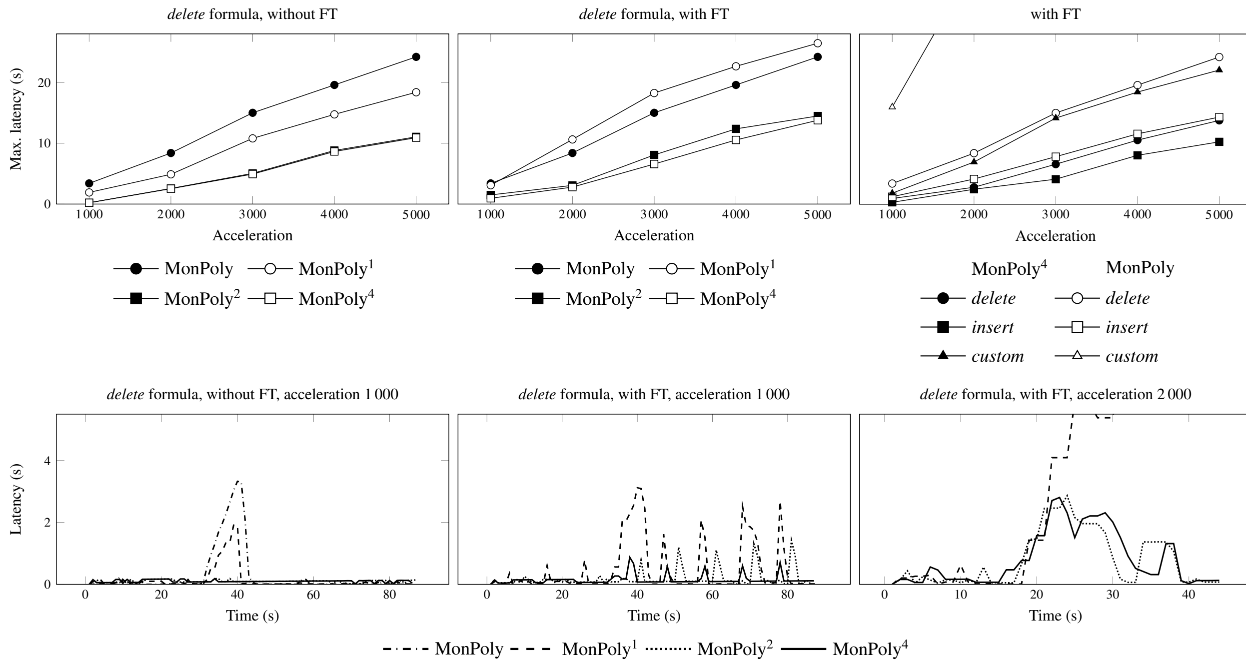


Fig. 9 *Nokia^MonPoly* experiments: monitoring the real-world stream with MonPoly

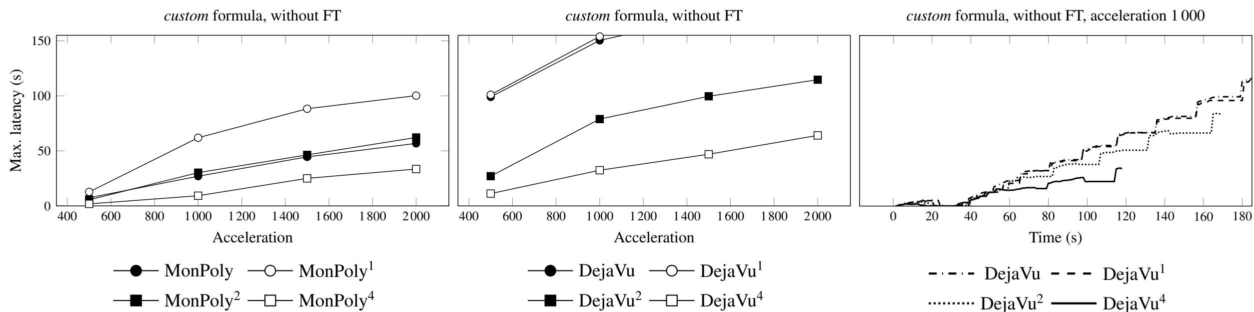
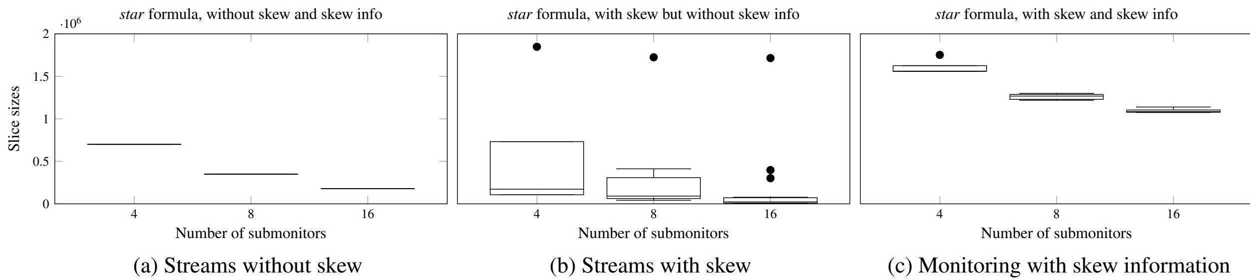


Fig. 10 *Nokia^DejaVu* experiments: monitoring the real-world stream with MonPoly and DejaVu

to monitor (top right plot), followed by the *delete*, and *insert* formulas, respectively. The other plots focus on the *delete* formula as it comes from the real use case and was not crafted by us. In contrast to the synthetic experiments, our framework's performance does not improve beyond 4 submonitors. However, if one considers the acceleration (up to 5 000) and the log's average event rate (110 events per second), our framework can process event rates higher than 500 000 events per second on average. At this point, the centralized parsing and slicing become the main performance bottleneck, which explains the marginal performance gains beyond 4 submonitors.

The top left and middle plots in Figure 9 contrast the performance overhead for fault tolerance (RQ_{ft}). The maximal latency is most visibly affected when the framework uses a single submonitor. The bottom three plots show how the latency changes over time during monitoring. These plots correspond to three individual runs while monitoring the *delete* formula. The leftmost plot shows the monitoring of the formula with respect to the stream sped up 1 000 times, with fault tolerance disabled. The middle and rightmost plots show runs with fault tolerance enabled for the accelerations of 1 000 and 2 000. The regularly occurring spikes in the latency graphs are caused by Flink's state snapshot algorithm, which is invoked every ten seconds.

Figure 10 compares the performance of our framework using MonPoly and DejaVu as submonitors when monitoring the *custom* formula on the log from the Nokia case study. Namely, MonPoly has lower maximum latency and in both cases our framework improves the latency (RQ_{gen}) when more submonitors are used. Figure 10's right-most plot shows how our framework improves DejaVu's current latency when monitoring the *custom* formula. The regular increases in la-



(a) Streams without skew (b) Streams with skew (c) Monitoring with skew information

Fig. 11 *Synthetic*$^{heavy\ hitters}$ experiments: impact of the skew and skew information on parallel monitoring

tency seen in each run are due to DejaVu's internal garbage collection, which tries to reduce its memory usage when storing previously seen parameter values [36].

Interestingly, using our framework with a single submonitor (MonPoly[1]) and without fault-tolerance lowers the maximum latency compared to a standalone run of MonPoly (top left plots in Figures 9 and 10). We conjecture that this results from the more efficient parsing and filtering of irrelevant events in our framework.

Finally, Figure 11a shows the number of events sent per submonitor when no skew is present in the stream. In the presence of skew, the event distribution is much less uniform (Figure 11b). When our framework is aware of the variables in the formula whose instantiations in the stream are skewed, it can balance the events evenly (Figure 11c), effectively reducing the maximum load of the submonitors (RQ_{skew}).

8 Conclusion and Future Work

Our work takes a substantial step towards efficient, parallel online monitoring of event streams with respect to policies written in expressive first-order languages. This entailed generalizing the offline slicing framework [11] to support online monitoring and the simultaneous slicing with respect to all free variables in the formula. Our work also builds a bridge to related research on query processing for databases and data streams. We adapted hash-based partitioning techniques from databases to obtain an automatic splitting strategy. We implemented a general approach to automatic slicing in Apache Flink and instantiated it with two existing tools for monitoring events with data, namely MonPoly and DejaVu. Our results demonstrate a significant performance improvement. For example, 16-fold parallelization allows us to increase the event rate from 10 000 to 75 000, while retaining sub-second maximum latency (Figure 7a).

In this article, we assumed that the stream's statistics are fixed. However, the automatic splitting strategy can be dynamically reconfigured by redistributing the submonitors' states coupled with the online collection of the statistics. We have already made some progress in implementing this extension and analyzing the tradeoff between the reconfiguration costs and the cost of using an imperfect splitting strategy [31, 53]. We also plan to refine our automatic splitting strategy to account explicitly for communication costs and to evaluate our approach in a distributed cluster. To achieve maximal scalability, it will be necessary to parallelize the splitter and to process events from multiple independent input streams [13].

References

1. The Nokia case study log file. `https://sourceforge.net/projects/monpoly/files/ldcc.tar/download` (2014)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Afrati, F.N., Joglekar, M.R., Ré, C., Salihoglu, S., Ullman, J.D.: GYM: A multiround distributed join algorithm. In: ICDT 2017, *LIPIcs*, vol. 68, pp. 4:1–4:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2017)
4. Afrati, F.N., Ullman, J.D.: Optimizing multiway joins in a map-reduce environment. IEEE Trans. Knowl. Data Eng. **23**(9), 1282–1298 (2011)
5. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The Stratosphere platform for big data analytics. VLDB J. **23**(6), 939–964 (2014)
6. Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.A., Hallé, S.: MapReduce for parallel trace validation of LTL properties. In: S. Qadeer, S. Tasiran (eds.) RV 2012, *LNCS*, vol. 7687, pp. 184–198. Springer (2012)
7. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: D. Giannakopoulou, D. Méry (eds.) FM 2012, *LNCS*, vol. 7436, pp. 68–84. Springer (2012)
8. Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: B. Bonakdarpour, S.A. Smolka (eds.) RV 2014, *LNCS*, vol. 8734, pp. 1–9. Springer (2014)
9. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: E. Bartocci, Y. Falcone (eds.) Lectures on Runtime Verification, *LNCS*, vol. 10457, pp. 1–33. Springer (2018)
10. Basin, D., Bhatt, B., Traytel, D.: Almost event-rate independent monitoring of metric temporal logic. In: A. Legay, T. Margaria (eds.) TACAS 2017, *LNCS*, vol. 10206, pp. 94–112. Springer (2017)
11. Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specifications. Form. Methods Syst. Des. **49**(1-2), 75–108 (2016)

12. Basin, D., Dardinier, T., Heimes, L., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: N. Peltier, V. Sofronie-Stokkermans (eds.) IJCAR 2020, *LNCS*, vol. 12166, pp. 432–453. Springer (2020)
13. Basin, D., Gras, M., Krstić, S., Schneider, J.: Scalable online monitoring of distributed systems. In: J. Deshmukh, D. Ničković (eds.) RV 2020, LNCS. Springer (2020). To appear.
14. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: Monitoring data usage in distributed systems. IEEE Trans. Software Eng. **39**(10), 1403–1426 (2013)
15. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
16. Basin, D., Klaedtke, F., Zălinescu, E.: The MonPoly monitoring tool. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 19–28. EasyChair (2017)
17. Basin, D., Krstić, S., Traytel, D.: Almost event-rate independent monitoring of metric dynamic logic. In: S. Lahiri, G. Reger (eds.) RV 2017, *LNCS*, vol. 10548, pp. 85–102. Springer (2017)
18. Bauer, A., Küster, J., Vegliach, G.: From propositional to first-order monitoring. In: A. Legay, S. Bensalem (eds.) RV 2013, *LNCS*, vol. 8174, pp. 59–75. Springer (2013)
19. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011)
20. Beame, P., Koutris, P., Suciu, D.: Skew in parallel query processing. In: R. Hull, M. Grohe (eds.) PODS 2014, pp. 212–223. ACM (2014)
21. Beame, P., Koutris, P., Suciu, D.: Communication steps for parallel query processing. J. ACM **64**(6), 40:1–40:58 (2017)
22. Bersani, M.M., Bianculli, D., Ghezzi, C., Krstić, S., Pietro, P.S.: Efficient large-scale trace checking using MapReduce. In: L.K. Dillon, W. Visser, L. Williams (eds.) ICSE 2016, pp. 888–898. ACM (2016)
23. Bianculli, D., Ghezzi, C., Krstić, S.: Trace checking of metric temporal logic with aggregating modalities using MapReduce. In: D. Giannakopoulou, G. Salaün (eds.) SEFM 2014, *LNCS*, vol. 8702, pp. 144–158. Springer (2014)
24. Bundala, D., Ouaknine, J.: On the complexity of temporal-logic path checking. In: J. Esparza, P. Fraigniaud, T. Husfeldt, E. Koutsoupias (eds.) ICALP 2014, *LNCS*, vol. 8573, pp. 86–97. Springer (2014)
25. Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., Tzoumas, K.: State management in Apache Flink®: Consistent stateful distributed stream processing. PVLDB **10**(12), 1718–1729 (2017)
26. Chothia, Z., Liagouris, J., Dimitrova, D.C., Roscoe, T.: Online reconstruction of structural information from datacenter logs. In: EuroSys 2017, pp. 344–358. ACM (2017)
27. Chu, S., Balazinska, M., Suciu, D.: From theory to practice: Efficient join query evaluation in a parallel database system. In: T.K. Sellis, S.B. Davidson, Z.G. Ives (eds.) SIGMOD 2015, pp. 63–78. ACM (2015)
28. Cormode, G., Hadjieleftheriou, M.: Methods for finding frequent items in data streams. VLDB J. **19**(1), 3–20 (2010)
29. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI 2004, pp. 137–150. USENIX Association (2004)
30. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: C. Colombo, M. Leucker (eds.) RV 2018, *LNCS*, vol. 11237, pp. 241–262. Springer (2018)
31. Fania, C.: Self-adaptive online monitoring. Bachelor's thesis, ETH Zürich (2019)
32. Feng, S., Lohrey, M., Quaas, K.: Path checking for MTL and TPTL over data words. Log. Methods Comput. Sci. **13**(3) (2017)
33. Ganguly, S., Silberschatz, A., Tsur, S.: Parallel bottom-up processing of Datalog queries. J. Log. Program. **14**(1&2), 101–126 (1992)
34. Hallé, S., Khoury, R.: Event stream processing with BeepBeep 3. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 81–88. EasyChair (2017)
35. Hallé, S., Khoury, R., Gaboury, S.: Event stream processing with multiple threads. In: S.K. Lahiri, G. Reger (eds.) RV 2017, *LNCS*, vol. 10548, pp. 359–369. Springer (2017)
36. Havelund, K., Peled, D.: Efficient runtime verification of first-order temporal properties. In: M. Gallardo, P. Merino (eds.) SPIN 2018, *LNCS*, vol. 10869, pp. 26–47. Springer (2018)
37. Havelund, K., Peled, D., Ulus, D.: First order temporal logic monitoring with BDDs. In: D. Stewart, G. Weissenbacher (eds.) FMCAD 2017, pp. 116–123. IEEE (2017)
38. Joglekar, M., Ré, C.: It's all a matter of degree – using degree information to optimize multiway joins. Theory Comput. Syst. **62**(4), 810–853 (2018)
39. Koutris, P., Beame, P., Suciu, D.: Worst-case optimal algorithms for parallel query processing. In: W. Martens, T. Zeume (eds.) ICDT 2016, *LIPIcs*, vol. 48, pp. 8:1–8:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2016)
40. Kuhtz, L., Finkbeiner, B.: LTL path checking is efficiently parallelizable. In: S. Albers, A. Marchetti-Spaccamela, Y. Matias, S.E. Nikoletseas, W. Thomas (eds.) ICALP 2009, *LNCS*, vol. 5556, pp. 235–246. Springer (2009)
41. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebr. Progr. **78**(5), 293–303 (2009)
42. Nagmote, S., Phadnis, P.: Massive scale data processing at Netflix using Flink. Flink Forward San Francisco 2019. `https://www.ververica.com/resources/flink-forward-san-francisco-2019/massive-scale-data-processing-at-netflix-using-flink` (2019). Accessed 31 May 2019
43. Nasir, M.A.U., Morales, G.D.F., García-Soriano, D., Kourtellis, N., Serafini, M.: The power of both choices: Practical load balancing for distributed stream processing engines. In: J. Gehrke, W. Lehner, K. Shim, S.K. Cha, G.M. Lohman (eds.) ICDE 2015, pp. 137–148. IEEE Computer Society (2015)
44. Nasir, M.A.U., Morales, G.D.F., Kourtellis, N., Serafini, M.: When two choices are not enough: Balancing at scale in distributed stream processing. In: ICDE 2016, pp. 589–600. IEEE Computer Society (2016)
45. Okcan, A., Riedewald, M.: Processing theta-joins using mapreduce. In: T.K. Sellis, R.J. Miller, A. Kementsietsidis, Y. Velegrakis (eds.) SIGMOD 2011, pp. 949–960. ACM (2011)
46. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: J. Misra, T. Nipkow, E. Sekerinski (eds.) FM 2006, *LNCS*, vol. 4085, pp. 573–586. Springer (2006)
47. Raszyk, M., Basin, D., Krstić, S., Traytel, D.: Multi-head monitoring of metric temporal logic. In: Y. Chen, C. Cheng, J. Esparza (eds.) ATVA 2019, *LNCS*, vol. 11781, pp. 151–170. Springer (2019)
48. Raszyk, M., Basin, D., Traytel, D.: Multi-head monitoring of metric dynamic logic. In: D.V. Hung, O. Sokolsky (eds.) ATVA 2020, *LNCS*, vol. 12302. Springer (2020). To appear.
49. Reger, G., Rydeheard, D.E.: From first-order temporal logic to parametric trace slicing. In: E. Bartocci, R. Majumdar (eds.) RV 2015, *LNCS*, vol. 9333, pp. 216–232. Springer (2015)
50. Rivetti, N., Querzoni, L., Anceaume, E., Busnel, Y., Sericola, B.: Efficient key grouping for near-optimal load balancing in stream processing systems. In: F. Eliassen, R. Vitenberg (eds.) DEBS 2015, pp. 80–91. ACM (2015)
51. Roşu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Log. Methods Comput. Sci. **8**(1) (2012)
52. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Scalable online first-order monitoring. In: C. Colombo, M. Leucker (eds.) RV 2018, *LNCS*, vol. 11237, pp. 353–371. Springer (2018)
53. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Adaptive online first-order monitoring. In: Y. Chen, C. Cheng, J. Esparza (eds.) ATVA 2019, *LNCS*, vol. 11781, pp. 133–150. Springer (2019)

54. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Implementation associated with this paper. https://bitbucket.org/krle/scalable-online-monitor (2019)
55. Schneider, J., Basin, D., Krstić, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: B. Finkbeiner, L. Mariani (eds.) RV 2019, *LNCS*, vol. 11757, pp. 310–328. Springer (2019)
56. Schneider, J., Traytel, D.: Formalization of a monitoring algorithm for metric first-order temporal logic. Archive of Formal Proofs (2019). https://devel.isa-afp.org/entries/MFOTL_Monitor.html. Entry point Slicing.thy.
57. Suri, S., Vassilvitskii, S.: Counting triangles and the curse of the last reducer. In: S. Srinivasan, K. Ramamritham, A. Kumar, M.P. Ravindra, E. Bertino, R. Kumar (eds.) WWW 2011, pp. 607–614. ACM (2011)
58. Vitorovic, A., Elseidy, M., Guliyev, K., Minh, K.V., Espino, D., Dashti, M., Klonatos, Y., Koch, C.: Squall: Scalable real-time analytics. PVLDB **9**(13), 1553–1556 (2016)