**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: a few thousands events per second. We 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 data 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 demonstrates a substantial scalability improvement for both tools.

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

## 1 Introduction

Large-scale software systems produce millions of log events per second [23, 37]. Identifying instances of interesting patterns in these high-volume, high-velocity data streams is a central challenge in the area of runtime verification and monitoring.

An *online monitor* takes a pattern, consumes a stream of data incrementally, and 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, current monitors are capable of handling hundreds of thousands of events per second in real time on commodity hardware [10, 15]. 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) [13], do not have this limitation. Various online monitors [7, 9, 13, 16, 32, 41, 43] 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 for a given 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. By being 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 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 offline monitoring (Section 2). Yet neither offline nor online monitoring is a data-parallel task in general. Thus, in

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).

Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
E-mail: {joshua.schneider, srdan.krstic, traytel}@inf.ethz.ch

some cases, monitors executing in parallel must synchronize. Alternatively, careful data duplication across these parallel 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 which parallel monitors the data should be sent to. It outputs 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 does not offer the expressiveness of 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 it to the offline setting.

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

- We generalize the offline slicing framework [11] to support simultaneous slicing on multiple variables and adapt the framework to the online setting (Section 4).
- We instantiate the slicing framework with an automatic splitting strategy (Section 5) inspired by hash-based partitioning and the hypercube algorithm [4, 34]. This algorithm has previously been used to parallelize relational join operators in databases. Our automatic strategy also addresses the imbalance of hash-based partitioning for skewed value distributions (short: *skew*) by separately handling events with frequently occurring values, which is 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 [13, 14] and DejaVu [32] 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 [12] and synthetic data exercising difficult cases (Section 7). We show that the overall parallel architecture has substantially improved throughput. While 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 hence improved monitoring performance.

An earlier version of this work was presented at RV 2018 [45]. 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), including a new subsection explaining the standard hypercube algorithm from databases (Section 5.1). Moreover, we demonstrate our framework's versatility by integrating DejaVu as a second black-box monitor in addition to MonPoly in our Apache Flink-based implementation (Section 6). Finally, we (re-)evaluate both versions of the resulting parallel online monitor (Section 7). For both, higher parallelism yields a 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 [44] and formalization [47] are publicly available. A separate paper reports on the related formal verification of an MFOTL monitor modeled after MonPoly [48].

## 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 implementation of the original framework limited it to the offline setting. In contrast, our Apache Flink implementation supports online monitoring. Finally, our implementation extends the framework with an automatic strategy selection that exhibits a balanced load distribution on the slices in our empirical evaluation.

Barre et al. [6], Bianculli et al. [20], and Bersani et al. [19] use task parallelism on different subformulas to parallelize propositional offline monitors. The degree of parallelization in these approaches is limited by the specification's syntactic complexity.

Parametric trace slicing [43] lifts propositional monitoring to parametric specifications. This algorithm takes a trace with parametric events and creates propositional slices with events grouped by their parameter instances, which can be independently monitored. 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 [41] 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 slices (in the domain's size) with grounded predicates, whereas we use as many slices as there are parallel monitors available.

Kuhtz and Finkbeiner [35] show that the LTL monitoring problem belongs to the complexity class $\mathrm{AC}^1(\mathrm{logDCFL})$ 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 the offline monitoring setting. A similar limitation applies to the work by Bundala and Ouaknine [21] and Feng et al. [27], 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. The publish-subscribe architectures used allow for ample parallelism. The languages used by CEP systems are often based on SQL extensions without a clear semantics. An exception is BeepBeep [29]: a multi-threaded [30] stream processor that supports LTL-FO$^+$, another first-order variant of LTL. The parallel computation in BeepBeep must, however, be scheduled 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 first proposed by Afrati and Ullman [4] in the context of MapReduce. The idea underlying the algorithm is similar to the algorithm by Suri and Vassilvitskii [49] used to count triangles and can be traced back to the parallel evaluation of datalog queries [28]. The hypercube algorithm was shown to be optimal for conjunctive queries with one communication round on skew-free databases [18], where skew refers to the distribution of data values.

The hypercube algorithm and other hash-based partitioning schemes are sensitive to skew. Rivetti et al. [42] suggest a greedy balancing strategy after identifying heavy hitters, which are frequently occurring input values. This approach is restricted to multi-way joins where all relations share a common join key. The algorithm by Joglekar et al. [33], which improves over hypercube using multiple communication rounds, is also restricted to multi-way joins. Nasir et al. [38, 39] balance skew for associative stream operators without explicitly identifying heavy hitters. Vitorovic et al. [50] combine the hash-based hypercube, prone to heavy hitters, with random partitioning [40], resilient to heavy hitters. Their combination only applies to multi-way joins 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 [34]. The heavy hitters must be known in advance in this approach. In contrast to the earlier algorithm by Beame et al. [17], 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) [13]. 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* $r(d_1, \ldots, d_{\iota(r)})$ is an element of $\mathbb{E} \times \mathbb{D}^*$. We call $1, \ldots, \iota(r)$ 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 $[a,b) := \{x \in \mathbb{N} \mid a \le x < b\}$, where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$ and $a < b$. Formulas $\varphi$ 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:

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

Along with the Boolean operators, MFOTL includes the metric past and future temporal operators $\bullet$ (*previous*), $\mathsf{S}$ (*since*), $\bigcirc$ (*next*), and $\mathsf{U}$ (*until*), which may be nested freely. We define other standard Boolean and temporal 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 $\varphi$.

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* $\rho$ is an infinite sequence $\langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$ of databases $D_i$ with associated time-stamps $\tau_i$. We assume discrete time-stamps, modeled as natural numbers $\tau \in \mathbb{N}$. We allow the event source to 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 ($\forall i.\ \tau_i \le \tau_{i+1}$) 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 $\varphi$ for a valuation $v$ at an index $i$ with respect to the stream $\rho = \langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$. The valuation $v$ is a mapping

| | |
|---|---|
| $v, i \models_\rho r(t_1, \ldots, t_n)$ | if $r(v(t_1), \ldots v(t_n)) \in D_i$ |
| $v, i \models_\rho t_1 \approx t_2$ | 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

$V(\varphi) \to \mathbb{D}$, assigning domain elements to the free variables of $\varphi$. 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 the argument $x$ 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 the prefix $\pi$ and $i < |\pi|$, $\pi[i]$ denotes $\pi$'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 generic characterization of an online monitor's input–output behavior. Slicing's fundamental property is that it preserves this behavior. 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.

### 4.1 Monitor Functions

Abstractly, a *monitor function* $\mathcal{M} \in \mathbb{P} \to \mathbb{O}$ maps stream prefixes to verdict outputs from a 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 input. The monitor may produce the verdicts incrementally, too. To represent this behavior at the level of monitor functions, we assume that verdicts are equipped with a partial order $\sqsubseteq$ indicating refinement and that a monitor function is a monotone map $\langle \mathbb{P}, \preceq \rangle \to \langle \mathbb{O}, \sqsubseteq \rangle$. This captures the intuition that as the monitor function receives more input, it produces more output, and (depending on the refinement ordering), does not retract previous verdicts.

We focus on *informative* monitor functions $\mathcal{M}_\varphi$, which output all satisfying valuations of the free variables of an MFOTL formula $\varphi$. An informative monitor's verdict is a set of tuples $(v, i)$, where $v$ is a valuation and $i$ is an index in the event stream. We call these tuples *satisfying valuations*. To detect violations of a safety property of the form $\Box \forall x_1 \ldots x_n.\, \varphi$, which is a standard application of monitors for runtime verification, one can monitor the negation $\neg\varphi$. Thus, the satisfying valuations in the monitor's verdict correspond to the violations, and they provide information about the values of the variables $x_1, \ldots, x_n$ that result in the violation.

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

$$\begin{aligned}
&\text{Soundness} && \forall \pi, v, i.\ (v, i) \in \mathcal{M}_\varphi(\pi) \\
&&& \longrightarrow i < |\pi| \wedge (\forall \rho \succeq \pi.\ v, i \models_\rho \varphi) \\
&\text{Completeness} && \forall \pi, \rho, v, i.\ \pi \preceq \rho \wedge i < |\pi| \wedge (\forall \rho' \succeq \pi.\ v, i \models_{\rho'} \varphi) \\
&&& \longrightarrow \exists \pi' \preceq \rho.\ (v, i) \in \mathcal{M}_\varphi(\pi')
\end{aligned}$$

By using the subset relation for the refinement order on verdicts, the granularity at which an online implementation can incrementally output each verdict is as a single satisfying valuation. 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 $\rho$ extending the prefix $\pi$. This property is sometimes called *impartiality* [36]. We restrict the output to indices covered by the prefix ($i < |\pi|$) because the output might be infinite otherwise. Our definition of completeness is a weak form of *anticipation*: Once a valuation $v$ is satisfied at an index $i < |\pi|$ on every possible extension of the prefix $\pi$, 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 [14] implements an informative monitor function for a practically relevant fragment of MFOTL [13]. Its output delay depends only on the future operators' intervals in the monitored formula. DejaVu [32] 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 indices at which violations had occurred. Since DejaVu does not support future temporal operators, its verdicts are never delayed.

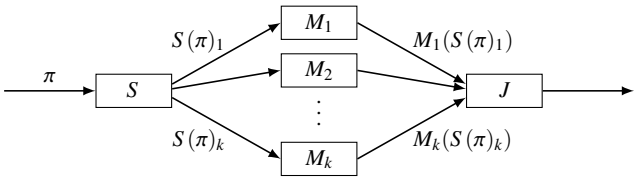



**Fig. 2** The parallelized monitor function $J\big(\lambda k.\ M_k(S(\pi)_k)\big)$, assuming $K = \{1,\ldots,k\}$

### 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 the 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 $\mathcal{M}$ 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 For all $\pi_1, \pi_2 \in \mathbb{P}$, $\pi_1 \preceq \pi_2$ implies $S(\pi_1)_k \preceq S(\pi_2)_k$, for all $k \in K$.

Correctness For all $\pi \in \mathbb{P}$, $J\big(\lambda k.\ M_k(S(\pi)_k)\big) = \mathcal{M}(\pi)$.

For an input prefix $\pi$, $S(\pi)$ denotes the collection of its slices. Each slice is identified by an element of $K$, 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 [41,43] can be seen as a particular slicer for monitor functions that arise from sliceable FO-LTL formulas [41, 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 $\pi_1 = \langle \tau_i, D_i \rangle_{i<|\pi_1|}$ and $\pi_2 = \langle \tau'_i, D'_i \rangle_{i<|\pi_2|}$, 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:

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

We also call a slicer with an event-separable splitter *event-separable*. We identify event-separable slicers $(K, M, S, J)$ 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 $\varphi$ at some index. We would like to evaluate $\varphi$ for each valuation to determine whether it is satisfied by that valuation. However, there are infinitely many valuations in the presence of infinite domains. The joint data slicer uses finitely many (possibly overlapping) slices, which taken together cover all possible valuations. For a given valuation, only a subset of the events is relevant to evaluate the formula. This allows us to exclude some events from each slice.

We assume without loss of generality that the bound variables in $\varphi$ are disjoint from the free variables $V(\varphi)$. Given an event $e = r(d_1,\ldots,d_n)$, the set $\mathit{matches}(\varphi, e)$ contains all valuations $v \in V(\varphi) \to \mathbb{D}$ for which there is a subformula $r(t_1,\ldots,t_n)$ in $\varphi$ where $v(t_i) = d_i$ for all $i \in \{1,\ldots,n\}$ with $t_i \in V(\varphi) \cup \mathbb{D}$.

**Definition 4** Let $\varphi$ 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 $\varphi$ with splitting strategy $f$ is the tuple $(K, \lambda k.\ \mathcal{M}_\varphi, \hat{S}_f, J_f)$, where

$$\hat{S}_f(e) = \bigcup\nolimits_{v \in \mathit{matches}(\varphi,e)} f(v),$$
$$J_f(s) = \bigcup\nolimits_{k \in K} (s_k \cap (\{v \mid k \in f(v)\} \times \mathbb{N})).$$

The following example demonstrates why the intersection with $\{v \mid k \in f(v)\} \times \mathbb{N}$ in the definition of $J_f$ is needed for some formulas, notably those involving equality.

*Example 1* Consider the formula $x \approx a \wedge \neg P(x)$, where $a$ is a constant. Even if a prefix contains $P(a)$, so that the formula is not satisfied at the corresponding index, the event will be omitted from all slices that do not have an associated valuation with $x = a$. If we do not filter the erroneous satisfying valuation from those slices, the result will be unsound.

We show next that $\mathcal{M}_\varphi^f(\pi) = J_f\big(\lambda k.\ \mathcal{M}_\varphi(S_f(\pi)_k)\big)$, the function that describes the behavior of the joint data slicer, is a monitor function, i.e., it is monotone, sound, and complete. As a first step, given a formula $\varphi$ and a set of valuations $R$, we define the formula's *relevant events* with respect to $R$ as $E_\varphi(R) = \{e \mid R \cap \mathit{matches}(\varphi, e) \neq \{\}\}$. 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 $\varphi$, a stream $\rho = \langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$, a set of valuations $R$, and a set of events $E$, with $E_\varphi(R) \subseteq E$. Let $\sigma = \langle \tau_i, D_i \cap E \rangle_{i \in \mathbb{N}}$. Then $v, i \models_\rho \varphi \longleftrightarrow v, i \models_\sigma \varphi$ for all $v \in R$ and $i \in \mathbb{N}$.*

*Proof* Proof by structural induction over the formula $\varphi$, 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 $\rho$ and $\sigma$.

$\varphi = r(t_1, \ldots, t_n)$: We have for any $v \in R$:

$$\begin{aligned} v, i \models_\rho \varphi &\longleftrightarrow r(v(t_1), \ldots, v(t_n)) \in D_i \\ &\overset{*}{\longleftrightarrow} r(v(t_1), \ldots, v(t_n)) \in D_i \cap E \longleftrightarrow v, i \models_\sigma \varphi. \end{aligned}$$

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

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

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

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

The step marked with $*$ is justified using the induction hypothesis for the formula $\psi$, 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: $S_f(\pi)_k[i] = (D_i \cap E_\varphi(\{v \mid k \in f(v)\}), \tau_i)$, for all $\pi = \langle \tau_i, D_i \rangle_{i < |\pi|}$ and $i < |\pi|$.

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

*Proof* The monotonicity of $\mathcal{M}_\varphi^f$ follows directly from $\mathcal{M}_\varphi$'s monotonicity. For soundness, fix $i$, $v$, and $\pi$ and assume $(v, i) \in \mathcal{M}_\varphi^f(\pi)$. Then, by $\mathcal{M}_\varphi^f$'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 $i < |S_f(\pi)_k| = |\pi|$ and $v, i \models_\sigma \varphi$ for all $\sigma \succeq S_f(\pi)_k$. Let $\rho$ be some stream extending $\pi$, 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 $i$, $v$, $\pi$ and $\rho = \langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$ and assume $i < |\pi|$, $\pi \preceq \rho$ (i.e., $\pi = \langle \tau_i, D_i \rangle_{i < |\pi|}$), and $\forall \rho' \succeq \pi.\ v, i \models_{\rho'} \varphi$. Let $k \in f(v)$ be some slice identifier. (Recall that $f(v)$ is nonempty.) We first show that for all $\sigma = \langle \tau_i, D_i' \rangle_{i \in \mathbb{N}}$ with $\sigma \succeq S_f(\pi)_k$ we have $v, i \models_\sigma \varphi$. For $i < |\pi|$, we have $D_i' = D_i \cap E_\varphi(\{v \mid k \in f(v)\})$ via $S_f$'s alternative characterization. Let the stream $\rho' = \langle \tau_i, E_i \rangle_{i \in \mathbb{N}}$, where $E_i = D_i$ for $i < |\pi|$ and $E_i = D_i'$ otherwise, and the stream $\sigma' = \langle \tau_i, E_i \cap E_\varphi(\{v \mid k \in f(v)\}) \rangle_{i \in \mathbb{N}}$. We calculate using Lemma 2 with $R = \{v \mid k \in f(v)\}$:

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

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 $\sigma'' = \langle \tau_i, D_i \cap E_\varphi(\{v \mid k \in f(v)\}) \rangle_{i \in \mathbb{N}}$, to obtain a $\pi''$ such that $\pi'' \preceq \sigma''$ and $(v, i) \in \mathcal{M}_\varphi(\pi'')$. Taking $\pi' = \langle \tau_i, D_i \rangle_{i < |\pi''|}$, we have $\pi' \preceq \rho$ and $\pi'' = S_f(\pi')_k$. By the definition of $\mathcal{M}_\varphi^f$, we conclude that $(v, i) \in \mathcal{M}_\varphi^f(\pi')$. □

The monitor functions $\mathcal{M}_\varphi$ and $\mathcal{M}_\varphi^f$ may differ. Both are informative, i.e., they produce correct verdicts (and eventually all verdicts by completeness) for the formula $\varphi$. Yet they may output verdicts with different delays. In general, the joint data slicer is only a slicer for $\mathcal{M}_\varphi^f$.

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

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

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

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

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

**Theorem 2** *Assume that the monitor function $\mathcal{M}_\varphi$ is sliceable. Then the joint data slicer $(K, \lambda k.\, \mathcal{M}_\varphi, \hat{S}_f, J_f)$ is a slicer for $\mathcal{M}_\varphi$.*

*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 easily from the definition of $\mathcal{M}^f_\varphi$, the sliceability assumption, and the fact that $f(v)$ is non-empty. □

*Example 2* Consider the formula $P(x,y) \wedge \neg \Diamond_{[0,5]}(P(y,x) \wedge Q(x))$. 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 $\pi = \langle(11, \{P(5,1), Q(2)\}), (12, \{P(5,7), Q(3), Q(5)\}), (21, \{P(7,5)\})\rangle$:

$$S_f(\pi)_1 = \langle(11, \{\}),\ (12, \{P(5,7), Q(5)\}),\ (21, \{P(7,5)\})\rangle$$
$$S_f(\pi)_2 = \langle(11, \{P(5,1), Q(2)\}),\ (12, \{P(5,7), Q(3)\}),\ (21, \{P(7,5)\})\rangle.$$

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 it will not receive any later event $Q(7)$, for which the second slice is responsible. This results in the spurious verdict $\langle x = 7, y = 5\rangle$, 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, which partition 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 $k$ satisfying $v(x) \in S_k$. 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 composition of 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 $(v(x) \bmod 2, v(y) \bmod 2)$, i.e., 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.

$$\begin{aligned}
\mathsf{mf}(r(t_1,\ldots,t_n)) &= \mathsf{true}\\
\mathsf{mf}(t_1 \approx t_2) &= (\exists a \in \mathbb{D}.\, t_1 = a \vee t_2 = a)\\
\mathsf{mf}(\neg(t_1 \approx t_2)) &= (\exists a, b \in \mathbb{D}.\, t_1 = a \wedge t_2 = b) \vee\\
&\quad\ (\exists x \in \mathbb{V}.\, t_1 = x \wedge t_2 = x)\\
\mathsf{mf}(\neg(\varphi \vee (\neg\psi))) &= ((\mathsf{mf}(\varphi) \wedge V(\varphi) \subseteq V(\psi)) \vee \mathsf{mf}^{\neg}(\varphi)) \wedge \mathsf{mf}(\psi)\\
\mathsf{mf}(\varphi \vee \psi) &= \mathsf{mf}(\varphi) \wedge \mathsf{mf}(\psi) \wedge V(\varphi) = V(\psi)\\
\mathsf{mf}(\exists x.\, \varphi) &= \mathsf{mf}(\varphi)\\
\mathsf{mf}(\bullet_I \varphi) &= \mathsf{mf}(\varphi)\\
\mathsf{mf}(\bigcirc_I \varphi) &= \mathsf{mf}(\varphi)\\
\mathsf{mf}(\varphi\, \mathsf{S}_I\, \psi) &= V(\varphi) \subseteq V(\psi) \wedge (\mathsf{mf}(\varphi) \vee \mathsf{mf}^{\neg}(\varphi)) \wedge \mathsf{mf}(\psi)\\
\mathsf{mf}(\varphi\, \mathsf{U}_I\, \psi) &= V(\varphi) \subseteq V(\psi) \wedge (\mathsf{mf}(\varphi) \vee \mathsf{mf}^{\neg}(\varphi)) \wedge \mathsf{mf}(\psi)
\end{aligned}$$

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

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 slicing strategy the intersection can be omitted, i.e., we can replace $J_f$ with $J'(s) = \bigcup_{k\in K} s_k$. We give a sufficient condition stemming from the following auxiliary 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 $\varphi$ be a formula and $R \neq \{\}$ a non-empty set of valuations. Assume that*

*(1) the formula $\varphi$ is* monitorable *[13, 48], i.e., satisfies the predicate* mf *defined in Figure 3. In particular, this requires that any negated subformula is guarded by a non-negated subformula, such that $\varphi$ can be monitored using finite relations [13];*
*(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 $\varphi$;*
*(3) no event name occurs twice in $\varphi$; 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 $v \in R$.*

*Moreover, assume $v, i \models_\rho \varphi$ for some $i \in \mathbb{N}$, a valuation $v$, and a stream $\rho = \langle\tau_i, D_i\rangle_{i\in\mathbb{N}}$ with $D_i \subseteq E_\varphi(R)$ for all $i \in \mathbb{N}$. Then $v \in R$.*

*Proof (Sketch)* By induction on the structure of monitorable formulas. The base cases are straightforward using the assumptions *(2)* and *(3)*. Note that monitorable formulas only allow negation to occur in formulas of the form $(\neg\varphi) \wedge \psi$ (i.e., $\neg(\varphi \vee (\neg\psi))$, $(\neg\varphi)\, \mathsf{S}_I\, \psi$, 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 $\psi$. This ensures that the satisfying valuations of these formulas are a subset of the satisfying valuations of $\psi$, allowing for a straightforward use of the induction hypothesis. The case $\varphi \wedge \psi$ (where both subformulas are not negated), requires joining the satisfying valuations of $\varphi$ and

$\psi$. Condition *(4)* makes sure that this join operation produces a valuation in $R$. □

The monitorability assumption is standard for monitors operating on finite tables. In fact, the MonPoly monitor only supports monitorable formulas [13]. In contrast, DejaVu supports non-monitorable formulas for the past-only non-metric fragment of MFOTL [32]. Observe that condition *(2)* of Lemma 3 rules out the formula from Example 1 and condition *(3)* rules out the formula from Example 7. We conclude this section with $J'$'s main property.

**Theorem 3** *Let $\varphi$ be a formula and $f$ the joint data slicer's slicing 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 $\varphi$ 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 $\rho = \langle \tau_i, D_i\rangle_{i\in\mathbb{N}}$ 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. In particular, computation costs should be evenly distributed, while overhead is kept low. Our approach to selecting automatically a suitable strategy is inspired by results from database theory and relies on basic stream statistics to optimize the submonitors' event rates, i.e., the number of events in a time period.

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. In online monitoring, the monitor's throughput must be high enough to process the incoming events with bounded delay, in particular if its buffering capacity is limited. A second, 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 achieve this, 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. We can also expect that 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: The Hypercube Algorithm

We recall the standard notion of full conjunctive queries [2], which represent a substantially less expressive language than MFOTL. However, the computational properties of conjunctive queries are well understood. In particular, researchers have devised and analyzed (near-)optimal distributed algorithms for computing conjunctive queries [3,4,17,18,33,34]. We focus on the hypercube algorithm [4,28,49] and discuss some previous results. The terminology used in this section 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 refer to the names $r \in \mathbb{E}$ also as *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. It 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

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

In the following, we assume that there is a linear ordering $x_1,\dots,x_k$ on the variables $V(q)$. Let $[n] \coloneqq \{0,\dots,n-1\}$ for $n \in \mathbb{N}$.

The basic hypercube algorithm [4] computes a full conjunctive query $q$ on a distributed, MapReduce-like system [26]. It is parametrized by the number $p$ of workers that participate in the computation, and a *share* $p_i \in \mathbb{N}$ for each variable $i \in \{1,\dots,k\}$ such that $\prod_i p_i = p$. Each worker is assigned a

unique coordinate vector $(x_1,\dots,x_k) \in [p_1] \times \cdots \times [p_k]$. 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 $i \in \{1,\dots,k\}$. The hash functions are known to all workers.
2. In the *map* phase, for every event $e$ in its local partition, each worker computes a set of workers $T(e)$, represented by their coordinates, to which it sends the event:

$$T(e) = \bigcup_{v \in matches(q,e)} \{(h_1(v(x_1)),\dots,h_k(v(x_k)))\}, \quad (1)$$

   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 have 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 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. [18] 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. 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. A database is skew-free if it does not contain *heavy hitters*, which are values whose degree with respect to some attribute $i$ and relation name $r$ exceeds $|D(r)|/p$. (Beame et al. [18], prove the bound for a more general notion of skew-free databases.)

Beame et al. [18] also show that linear programming can be used to optimize the shares. The input to the optimization is the full conjunctive query and the relation sizes of the database on which the query should be computed. With the optimized shares, the hypercube algorithm matches the lower bound with respect to the maximum load for computing a full conjunctive query with a single round of computation. The lower bound holds in the *Massively Parallel Communication (MPC)* model, in which the $p$ workers can communicate arbitrary messages over private channels and have unbounded computational power. A single round means that there is one synchronization step after the initial communication phase. Afterwards, each worker computes only locally. However, optimizing the shares with linear programming does not yield integer values in general. As an alternative, Chu et al. [24] 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 [18])* 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 $p_2 = p_3 = 1$, which 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 workers $T(e)$ |
|---|---|
| $P(0,1)$ | 010, 011, 012, 013 |
| $P(1,1)$ | 110, 111, 112, 113 |

| event $e$ | containing slices $T(e)$ |
|---|---|
| $Q(1,7)$ | 013, 113, 213, 313 |
| $R(7,0)$ | 003, 013, 023, 033 |

The events $P(0,1)$, $Q(1,7)$, and $R(7,0)$ are sent to the worker with coordinates 013, which 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. [34] propose an extension that is worst-case optimal also for skewed databases. They assume that all heavy hitters in the database are known in addition to the relation sizes. In this extension, which we simply call *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 hypercube instances 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 contains some heavy hitters. We analyze the optimal shares for the triangle query $q_T$ and the variable sets $H_1 = \{x_1\}$, $H_2 = \{x_1, x_2\}$, and $H_3 = \{x_1, x_2, x_3\}$. If no variable has a heavy hitter ($H = \{\}$), the shares from Example 4 apply. The remaining variable sets have symmetric solutions. In the case $H_1$, the optimal shares are $p_{H_1,1} = 1$ and $p_{H_1,2} = p_{H_1,3} = p^{1/2}$. Each worker then receives at most $1/p^{1/2}$ of the events for which only $x$ is assigned a heavy hitter. In the case $H_2$, the optimal shares are $p_{H_2,1} = p_{H_2,2} = 1$ and $p_{H_2,3} = p$, so at most $1/p$ of the corresponding events are sent to the workers. Finally in the case $H_3$, 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 $H_3$ 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

Since we aim to adapt the hypercube algorithm, we generalize the notions of relation size and heavy hitters to event streams. The automatic slicer takes these statistics as inputs to optimize the strategy accordingly. Since streams are unbounded, we consider non-overlapping time intervals of a fixed size $\Delta$. 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 is a parameter of our model. Choosing a larger $\Delta$ smoothes out transient irregularities in the stream and thus reduces the variability of the characteristics. The downside is lower precision, which may impact monitoring latency: Assume 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 $\Delta$ is, the larger is the difference between this maximal latency and the best case.

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

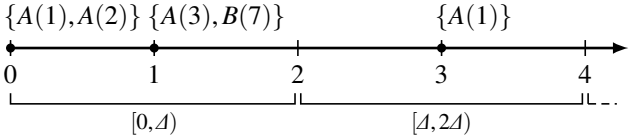
$$\gamma_\theta(r) = \frac{1}{\Delta} \cdot \sum_{\tau_i/\Delta \in [\theta, \theta+1)} |D_i(r)|.$$

As before, $D_i(r)$ denotes the set of events with the name $r$ in the database $D_i$. The *event rate* at time $\theta$ 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 $i \in \{1, \ldots, \iota(r)\}$, the *frequency* $F_\theta(d, r, i)$ of $d \in \mathbb{D}$ is

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

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 $\theta$ 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 $\Delta = 2$ 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 $\gamma_\theta$, i.e.,

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

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 $\gamma_\theta$, 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 use this problem statement to guide 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 $t$). Equivalently, the choice of the splitting strategy applies to a single interval of size $\Delta$. We therefore omit the index $\theta$ and write $\gamma(r)$, $\lambda(f)$, and so forth. We have started to address time-varying statistics in a separate work [46].

### 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 operator 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 defined in (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 $\xi_H : [p_{H,1}] \times \cdots \times [p_{H,k}] \to K$. We can then describe the mapping phase of the extended algorithm by the single equation

$$T'(e) = \bigcup\nolimits_{v \in matches(q,e)} \{\xi_H(h_{H,1}(v(x_1)), \ldots, h_{H,k}(v(x_k))) \mid H = heavy(q,v)\},$$

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 $\varphi$, respectively, may depend on $e$.

To complete the transition from queries to MFOTL formulas, we determine the equivalent of $heavy(q,v)$ for $\varphi$. 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 $heavy_1(\varphi,x)$ be the union of all $\mathcal{H}(r,i)$ for which there is a subformula $r(y_1,\ldots,y_{\iota(r)})$ in $\varphi$ with $y_i = x$. We then define $heavy(\varphi,v)$ as $\{x \mid v(x) \in heavy_1(\varphi,x)\}$.

The set

$$f(v) = \{\xi_H(h_{H,1}(v(x_1)), \ldots, h_{H,k}(v(x_k))) \mid H = heavy(\varphi,v)\}$$

is nonempty and thus a valid splitting strategy. We call $f$ the *hypercube strategy* for $\varphi$ with parameters $h_{H,j}$, $\xi_H$, and $\mathcal{H}$. Algorithm 1 computes $f$ for a given event, where the partial map $\langle\rangle$ is undefined everywhere, and $\mathrm{codom}(h)$ denotes $h$'s codomain. The algorithm first computes a partial valuation

**Algorithm 1:** Hypercube Strategy

**Data:** $\varphi$ with free variables $x_1,\ldots,x_n$; $\langle h_{H,i}\rangle_{H,i}$, $\langle \xi_H\rangle_H$, $\langle heavy_1(\varphi,x_i)\rangle_i$; event $e = r(d_1,\ldots,d_{\iota(r)})$
**Result:** slice identifiers $T = \bigcup_{v \in matches(\varphi,e)} f(v)$

```
1  T ← {};
2  foreach subformula r'(y_1,...,y_ι(r)) of φ do
3      if r = r' then // compute partial valuation v induced by e
4          v ← ⟨⟩;
5          for i ← 1 to ι(r) do
6              if y_i ∈ 𝕍 then
7                  if v ≠ ⊥ ∧ (v(y_i) = ⊥ ∨ v(y_i) = d_i) then
                       v ← v[y_i ↦ d_i] else v ← ⊥;
8              else if y_i ≠ d_i then
9                  v ← ⊥;
10             end
11         end
12         if v ≠ ⊥ then // recursively enumerate slices for each
             heavy(φ,v') where v' extends v
13             T ← T ∪ 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, ⟨⟩, 1)
20     else if v(x_i) = ⊥ ∧ heavy_1(φ,x_i) ≠ {} then // unknown if
         x_i is a heavy hitter because it is not assigned
21         return AllHeavy(v, H, i+1) ∪ AllHeavy(v, H ∪ {x_i},
             i+1)
22     else if v(x_i) ∈ heavy_1(φ,x_i) then
23         return AllHeavy(v, H ∪ {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 {ξ_H(t)}
31     else if v(x_i) = ⊥ then    // x_i not assigned: consider all
         coordinates
32         return ⋃_{z ∈ codom(h_{H,i})} AllExtensions(v, H, t[i ↦ z],
             i+1)
33     else
34         return AllExtensions(v, H, t[i ↦ h_{H,i}(v(x_i))], i+1)
35     end
36 end
```

$v$ for each of the formula's predicates by matching the event with the predicate. The valuation assigns values to those variables that occur in the predicate. It then 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 $heavy_1(\varphi,x) = \{\}$ because there is no valuation $v'$ where $H = heavy(\varphi,v')$. Then, for each $H$, the algorithm constructs the finite set of

coordinates $(h_{H,1}(v'(x_1)),\dots,h_{H,n}(v'(x_n)))$ 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 $\xi_H$. As with databases, we select $h_{H,j}$ uniformly at random with a given codomain $[p_{H,j}]$. The *shares* $p_{H,j}$ 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 $\xi_H$ can be arbitrary; in practice, we map coordinates to slice identifiers in $[p]$:

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

*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 $\varphi = P(x_1) \wedge \bullet P(x_2)$ 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 $\varphi = ((\blacklozenge_{[0,10]} P(x_1,x_2)) \wedge Q(x_2,x_3)) \wedge \neg \Diamond_{[0,10]} R(x_3,x_1)$ 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 $\varphi$ 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 $\xi_H$, 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 $heavy_1(\varphi,x)$ 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 in either case. 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 an 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 $\varphi = P(x_1) \wedge Q(x_1,x_2)$, $p_{\{\},1} = p_{\{x_2\},1} = 2$, 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. [18, Theorem 3.2] and Koutris et al. [34, Theorem 2] from the database setting to ours. This allows us to use the algorithm of Chu et al. [24] to optimize the shares. The transfer is based on the simple observation that applying the hypercube strategy to an event stream incurs the same load as using the hypercube algorithm with a certain database. If we focus on an interval, this database is the (multiset) union of all databases in the stream that belong to the interval.[1] 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 $r(y_1,\dots,y_{\iota(r)}) \leq \varphi$ denote the fact that $r(y_1,\dots,y_{\iota(r)})$ is a subformula of $\varphi$. The maximum load $\lambda$ is bounded from above by

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

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

Algorithm 2 optimizes the shares and selects the hash functions. For each $H \subseteq V(\varphi)$, it first iterates over all valid share vectors $p_H = (p_{H,1},\dots,p_{H,n})$. A share vector is valid if $\prod_{1\leq i\leq n} p_{H,i} \leq p$, 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 [24].

[1] We need to use multisets because events may be repeated at different indices. The upper bound from Beame et al. [18] extends to multisets.

**Algorithm 2:** Hypercube Optimization

**Data:** $\varphi$ with free variables $x_1, \ldots, x_n$; number of submonitors $p$, relative event rates $\langle \gamma'(r) \rangle_r$
**Result:** parameters $\langle h_{H,i} \rangle_{H,i}$ 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
```

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. [24] 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.

### 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 $\varphi$ and $n$ is the number of free variables in $\varphi$. We assume $n, p \geq 1$ and that all operations that involve $\mathbb{D}$ and slice identifiers in $[p]$ are $O(1)$, 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 might be 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 lexicographically sorted order instead (lines 5–10). We do not show this optimization to maintain clarity. Note that Algorithm 2 needs to be run only once when the monitor is initialized, while Algorithm 1 must be invoked for every event.

The minimum possible load achieved by the hypercube strategy depends on the pattern of free variables in the formula's atoms. A detailed discussion is provided by Koutris et al. [34]. 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. 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 arbitrary MFOTL formulas. This already becomes evident for simple non-temporal formulas with disjunctions, e.g., $P(x_1, x_2) \wedge (Q(x_1) \vee Q(x_2))$. 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 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. It is not clear how this technique can be generalized to formulas, in particular with nesting inside of 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 some aspects of the formulas' syntactic structure.

We assumed that the submonitors' throughput does not depend on the events themselves. It was therefore sufficient to minimize their load to optimize the throughput. This simplification is not always accurate for monitors such as MonPoly.

The reason is that MonPoly constructs intermediate results, whose size impacts the complexity of further operations inside the monitor. The size of results depends in turn on the monitor's input. 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.

Stream statistics such as $\gamma(r)$ and $\mathcal{H}(r,i)$ can be obtained efficiently while processing an online stream, for example using approximate algorithms [25]. However, our approach assumes that the statistics are available before the start of monitoring, as they are used to optimize the splitting strategy. Moreover, the statistics may change over time. A possible 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 being 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 adaptive slicing [46].

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 in the monitoring algorithm. 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 monitor based on the joint data slicer, building on the Apache Flink stream processing framework. The source code consists of approximately 3,100 lines of Java and Scala and is publicly available [44]. Given a formula, the monitor 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 monitoring of the slices is delegated to an external tool. The implementation currently supports both the tools MonPoly [14] and DejaVu [32].

The Flink API enables the construction of dataflow graphs with stream operators as nodes. 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 [22]: 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 parallel online monitor are the formula, the number of parallel monitors, the stream statistics for the shares' optimization, and the heavy hitter values. The monitor precomputes the shares using Algorithm 2 and outputs a dataflow graph, shown in Figure 4. 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 first 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 precomputed optimized shares and heavy 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 the definition of $J_f$ (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 to the Flink pipeline to prevent 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 the external state. We have extended MonPoly with control commands that implement this interface. 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

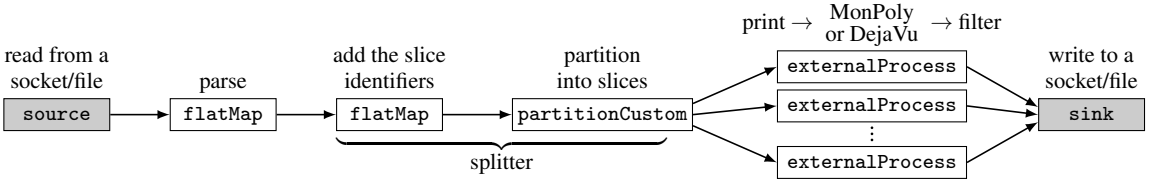



**Fig. 4** Our parallel online monitor's dataflow graph

temporary file. The part of the monitor's output received after the checkpoint instruction's arrival at the `externalProcess` operator is 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. Therefore, the monitor is currently not fault-tolerant if DejaVu is used. We conjecture that extending DejaVu with the state interface is possible with modest effort.

DejaVu monitors closed formulas only and reports violating instead of satisfying valuations. Therefore, we first close the parallel monitor's input formula $\varphi$ by adding a prefix of existential quantifiers. We then negate the closed formula before we pass it to DejaVu, such that its output is consistent with MonPoly's output. The splitter uses the original formula $\varphi$ because it is only effective if there are free variables. As the output of DejaVu consists only of the violating indices for the closed formula, we cannot compute the intersection from $J_f$'s definition with $\varphi$'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 to DejaVu to certain formulas, and we cannot take heavy hitters into account 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 have to ensure that events arrive at the submonitors in chronological order. This order is no longer guaranteed if the splitter is partitioned into concurrent tasks. A possible solution is to buffer and reorder events at each submonitor. We leave the analysis of such an extension for future work.

## 7 Evaluation

To assess the scalability, effectiveness, and generality of our framework, we structure our evaluation to answer the following specific research questions.

RQ1: How does our monitor scale with respect to different formulas?
RQ2: How does our monitor scale with respect to the index rate and the event rate?
RQ3: Can knowledge about the frequency of event names improve performance?
RQ4: Can knowledge about heavy hitter values improve performance?
RQ5: Can our monitor handle data from real-world online settings?
RQ6: How much overhead is incurred by using our framework? Specifically, how does it compare to the standalone tools MonPoly and DejaVu?
RQ7: How much overhead is incurred by supporting fault tolerance (FT)?
RQ8: Is our framework general (i.e., does it also scale when using a different submonitor)?

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 our framework's ability to scalably solve realistic problems. Figure 5 summarizes the parameters used for each experiment, which we explain next.

*Synthetic Experiments* For the experiments with synthetic streams (the $Synthetic^{MonPoly}$ experiments in Figure 5) we monitor three formulas: *star*, *linear*, and *triangle*, shown in Figure 6. Different occurrence patterns of free variables in the formulas are used to test RQ1. The formulas cover common patterns in database queries [18], which we additionally extend with temporal operators. We have implemented a stream generator tailored to each of the three formulas. It takes a random seed and synthesizes streams with configurable characteristics. Specifically, the synthesized streams on average have constant characteristics across all time indices $\theta$. 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 RQ2. Figure 5 summarizes the event rates used in our experiments. Note that we evaluate only those combinations of event rates and number of processors 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

| Experiments | Synthetic streams: $Synthetic^{MonPoly}$ | $Synthetic^{DejaVu}$ | $Synthetic^{heavy\ hitters}$ | Real-world streams: $Nokia^{MonPoly}$ | $Nokia^{DejaVu}$ |
|---|---|---|---|---|---|
| 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* |
| Processors | 1, 4, 8, 16 | 1, 4, 8, 16 | 4, 8, 16 | 1, 2, 4, 8 | 1, 2, 4, 8 |
| Event rate | 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 rate | 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 distribution | 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 | event rate × 60 | event rate × 60 | 9.5 million | 9.5 million |
| Accelerations | 1 | 1 | 1 | 1k, 2k, 3k, 4k, 5k | 500, 1k, 1.5k, 2k |
| Stage | online, offline | online, offline | offline | online | online |
| Fault tolerance | both | no | no | both | no |

**Fig. 5** Summary of parameters used in our experiments

with the standalone MonPoly instance on streams with event rates up to 20 000 (denoted as 20k in Figure 5).

To test RQ3 and RQ4, 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 using 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 (see Figure 5), we additionally generate streams with heavy hitter values in valuations of variable $a$ in the *star* formula and variable $b$ in *linear* and *triangle* formulas, with their Zipf exponents set to 2.

*Real-world Experiments* To test RQ5, we use logs from Nokia's Data Collection Campaign [12]. The campaign collected data from the mobile phones of 180 participants and propagated the data through three databases, `db1`, `db2`, and `db3`. The phones uploaded their data directly to `db1`, while a synchronization script `script1` periodically copied the data from `db1` to `db2`. Then, `db2`'s triggers anonymized and copied the data to `db3`. The participants could query and delete their own data stored in `db1`. Deletions were propagated to all databases.

To obtain streams suitable for online monitoring, we have developed a tool 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 to evaluate 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 our replayer tool to accelerate the fragment up to 5 000 times. The fragment contains roughly 9.5 million events and has an average event rate of 110 events per second. Combined with 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 [44] are publicly available.

We monitor formulas *insert*, *delete* (Figure 6), and *custom* (Figure 7). Formulas *insert* and *delete* come from the Nokia's Data Collection Campaign and have been shown to be challenging to monitor [12]. We used our knowledge of the data set to additionally craft the *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 in order for it to produce 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 start the monitor with the stored state as its initial 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 in order to answer RQ3 and RQ4.

*Monitors* To test RQ6 and RQ8, we use MonPoly and DejaVu as standalone monitors, and also as parallel submonitors within our framework. To accommodate DejaVu, which implements a slightly different monitor function than MonPoly, we need to adapt the parameters of our two families of exper-

$$
\begin{array}{lcl}
\mathit{star} & = & \big((\blacklozenge_{[0,10\,\mathrm{s})}\, P(a,b)) \wedge Q(a,c)\big) \wedge \lozenge_{[0,10\,\mathrm{s})}\, R(a,d) \\
\mathit{linear} & = & \big((\blacklozenge_{[0,10\,\mathrm{s})}\, P(a,b)) \wedge Q(b,c)\big) \wedge \lozenge_{[0,10\,\mathrm{s})}\, R(c,d) \\
\mathit{triangle} & = & \big((\blacklozenge_{[0,10\,\mathrm{s})}\, P(a,b)) \wedge Q(b,c)\big) \wedge \lozenge_{[0,10\,\mathrm{s})}\, R(c,a) \\
\mathit{insert} & = & (\mathit{insert}(u,\mathsf{db1},\mathit{pid},\mathit{dt}) \wedge \mathit{dt} \not\approx \mathsf{unknown}) \wedge \big(\neg \lozenge_{[0,30h]}\, \exists u'.\, (\mathit{insert}(u',\mathsf{db2},\mathit{pid},\mathit{dt}) \vee \mathit{delete}(u',\mathsf{db1},\mathit{pid},\mathit{dt}))\big) \\
\mathit{delete} & = & \big((\mathit{delete}(u,\mathsf{db1},\mathit{pid},\mathit{dt}) \wedge \mathit{dt} \not\approx \mathsf{unknown}) \wedge (\neg \blacklozenge_{[0,30h)}\, \exists u', p'.\, \mathit{insert}(u',\mathsf{db1},p',\mathit{dt}))\big) \vee \\
 & & \big((\mathit{delete}(u,\mathsf{db1},\mathit{pid},\mathit{dt}) \wedge \mathit{dt} \not\approx \mathsf{unknown}) \wedge ((\blacklozenge_{[0,30h)}\, \exists u', p'.\, \mathit{insert}(u',\mathsf{db1},p',\mathit{dt})) \vee \lozenge_{[0,30h)}\, \exists u', p'.\, \mathit{insert}(u',\mathsf{db2},p',\mathit{dt}))\big)\big) \wedge \\
 & & (\neg \lozenge_{[0,30h)}\, \exists u', p'.\, \mathit{delete}(u',\mathsf{db2},p',\mathit{dt}))
\end{array}
$$

**Fig. 6** MFOTL formulas used in the evaluation

$$
\begin{array}{lcl}
\mathit{star\text{-}past} & = & \big(\blacklozenge_{[0,\infty)}((\blacklozenge_{[0,\infty)}\, P(a,b)) \wedge Q(a,c))\big) \wedge R(a,d) \\
\mathit{triangle\text{-}past} & = & \big(\blacklozenge_{[0,\infty)}((\blacklozenge_{[0,\infty)}\, P(a,b)) \wedge Q(b,c))\big) \wedge R(c,a)
\end{array}
\qquad
\begin{array}{lcl}
\mathit{linear\text{-}past} & = & \big(\blacklozenge_{[0,\infty)}((\blacklozenge_{[0,\infty)}\, P(a,b)) \wedge Q(b,c))\big) \wedge R(c,d) \\
\mathit{custom} & = & \exists u_1\, db_1.\, \mathit{select}(u_1,db_1,\mathit{pid}_1,\mathit{dt}) \wedge \\
 & & \big(\blacklozenge_{[0,\infty)}\, \exists u_2\, db_2.\, \mathit{insert}(u_2,db_2,\mathit{pid}_2,\mathit{dt})\big)
\end{array}
$$

**Fig. 7** Past-only non-metric MFOTL formulas used in the evaluation with DejaVu

iments (see the $\mathit{Synthetic}^{\mathit{DejaVu}}$ and $\mathit{Nokia}^{\mathit{DejaVu}}$ experiments in Figure 5). First, we use the four formulas shown in Figure 7, 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. Note that the verdicts of the formulas in Figure 7 are not affected by this transformation.

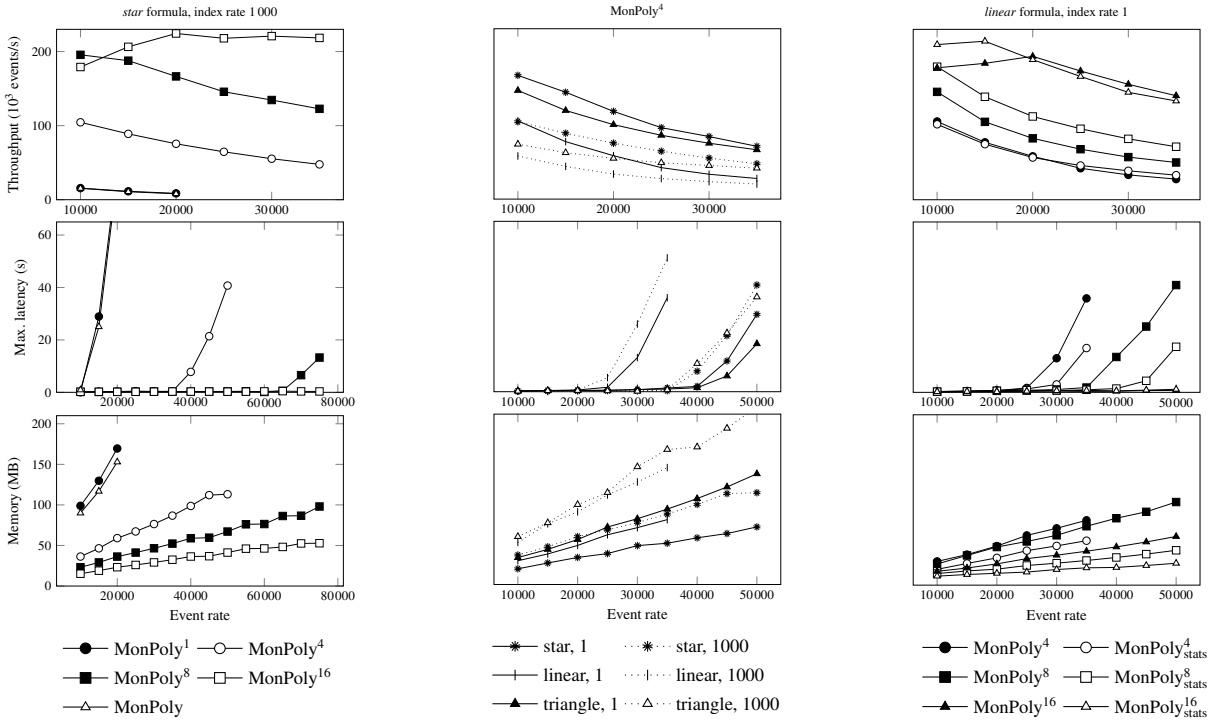
Additionally, we run the experiments both with and without Flink's fault tolerance mechanism to attest its impact on performance (RQ7). This is only done when our framework uses MonPoly as a submonitor, since DejaVu does not support state checkpointing.

*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. In order to assess the scalability of our framework, 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 UNIX `time` command to measure total execution time, i.e., the time between the moment when the replayer tool 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 tool 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 tool 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.

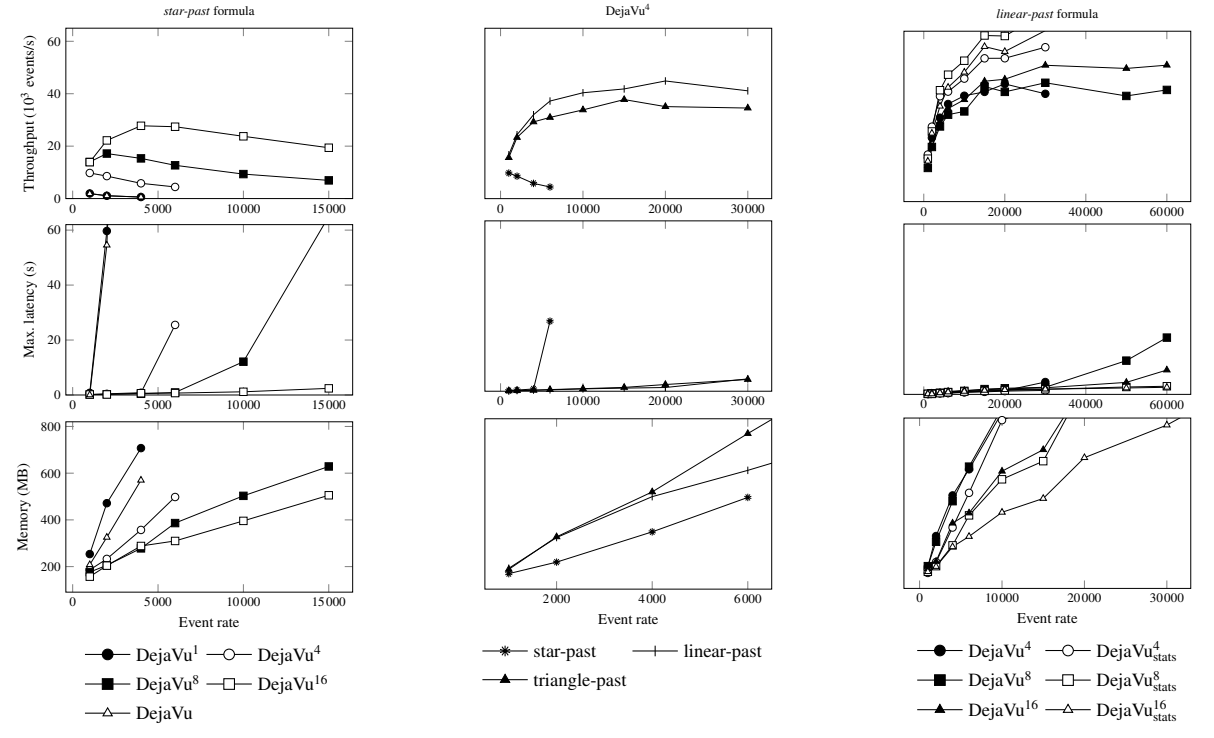
When the latency is higher than one second, the latency marker gets delayed too and a timely value cannot be produced. Flink reports zeros for the current latency in this case, while we consider the latest non-zero value. This significantly reduces the noise in our measurements. Flink also measures the number of events each submonitor receives. In addition to *online* monitoring experiments, where we use our replayer tool 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 monitored events and the measured offline execution time. Since we focus on performance measurements, we discard the tool's output during the 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 8 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 $\mathrm{Tool}^N$ denote




(a) Different cores (b) Different formulas and index rate (c) Using stream statistics

**Fig. 8** *Synthetic*$^{MonPoly}$ experiments: monitoring synthetic streams with MonPoly and with fault tolerance




(a) Different parallelism (b) Different formulas (c) Using stream statistics

**Fig. 9** *Synthetic*$^{DejaVu}$ experiments: monitoring synthetic streams with DejaVu and without fault tolerance

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 (RQ6): A standalone run of a Tool exhibits the same performance as a run of our framework with one submonitor (Tool$^1$).

Figure 8a 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. With an event rate of 15 000 events per second, our tool already exhibits 27 second latency 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 RQ2: our tool handles significantly higher event rates by using more parallel submonitors.

Figure 8b shows achieved throughput (top), maximum latency (middle), and maximal memory consumption (bottom) of our tool when monitoring *star*, *triangle*, and *linear* formulas using 4 submonitors. The plots show six graphs, where a graph shows 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 [13], the overall framework is also affected (RQ2). The event rate gain due to parallelized monitoring depends on the variable occurrence patterns in the monitored formula (RQ1). 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 8c positively answers RQ3 by showing that our tool's performance increases independently of the number of submonitors when such statistics about the stream are known in advance. We use $\text{Tool}^N_{\text{stats}}$ to denote that our framework runs Tool on $N$ submonitors with relative event rates provided ahead of time.

Figure 9 shows the results of the same experiments as in Figure 8 but now using our framework with DejaVu as the submonitor. In these experiments, fault tolerance was not enabled. Similarly as before, the experiments show that our framework can handle higher event rates by using more parallel submonitors (RQ2). Regarding RQ8, they demonstrate improved throughput, latency, and memory consumption also with a different first-order monitor, namely DejaVu. Note that both Figure 8a and Figure 9a answer RQ6: They show that our framework achieves better performance than MonPoly and DejaVu on their own, except when the framework uses a single submonitor, where it exhibits essentially the same performance.

Figure 10 summarizes the results of using our framework with MonPoly to monitor the real-world log from the Nokia case study. The event and index rates are defined by the log; we only control the acceleration used by the replayer tool. In this experiment, our framework's performance does not improve beyond 4 submonitors, since for event rates higher than 500 000 events per second, the centralized parsing and slicing becomes the main performance bottleneck. The top left and middle plots contrast the performance overhead for fault tolerance (RQ7). 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 regular spikes in the latency graphs stem from Flink's state snapshot algorithm.

Figure 11 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, using MonPoly yields lower maximum latency and in both cases our framework improves the latency (RQ8) when more submonitors are used. Figure 11's right-most plot shows how our framework improves DejaVu's current latency when monitoring the *custom* formula. The regular increases in latency 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 [31].

Finally, Figure 12a 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 12b). However, when our monitor is aware of the variables in the formula whose instantiations in the stream are skewed, it can balance the events evenly (Figure 12c), effectively reducing the maximum load of the submonitors (RQ4).

## 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

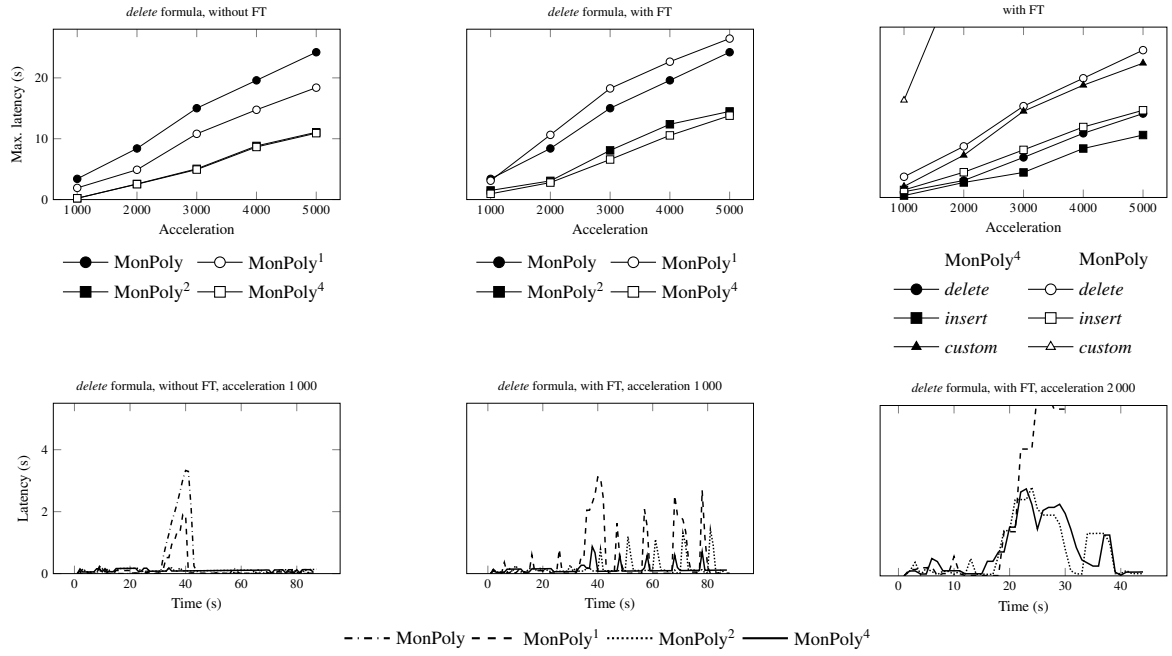



**Fig. 10** *Nokia*$^{MonPoly}$ experiments: monitoring the real-world stream with MonPoly

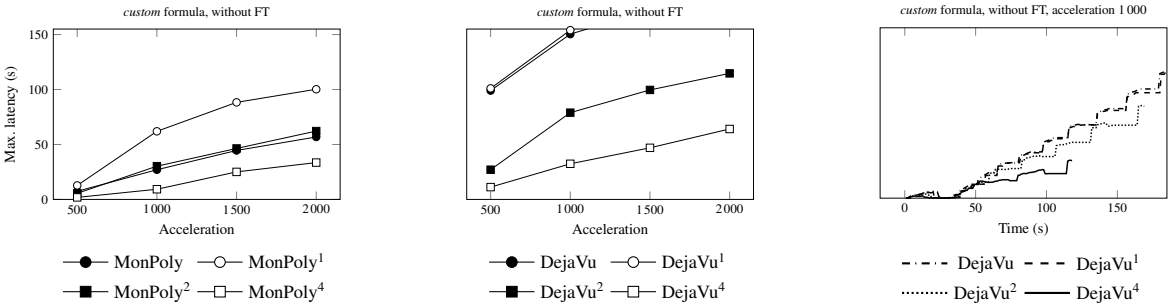


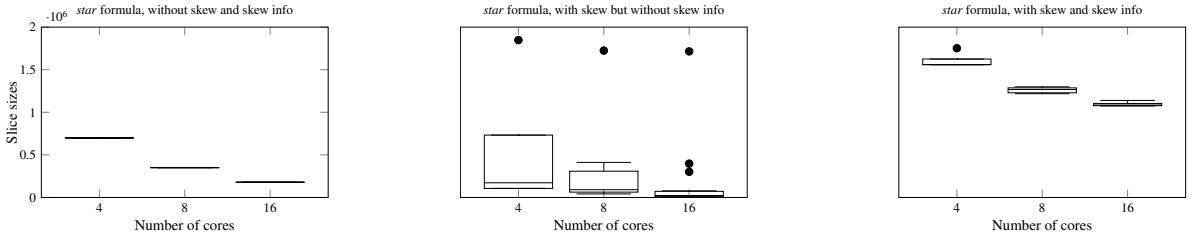

**Fig. 11** *Nokia*$^{DejaVu}$ experiments: monitoring the real-world stream with MonPoly and DejaVu

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, while retaining sub-second latency, 16-fold parallelization allows us to increase the event rate from 10 000 to 75 000 (Figure 8a).

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 an 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 [46]. We also plan to refine our automatic splitting strategy to account explicitly for communication costs and to evaluate our approach on a distributed computing cluster. To achieve maximal scalability, it will be necessary to parallelize the splitter and to process events from multiple independent input streams.

## 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.,




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

**Fig. 12** *Synthetic*$^{heavy\ hitters}$ experiments: impact of the skew and skew information on parallel monitoring

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., Harvan, M., Klaedtke, F., Zălinescu, E.: Monitoring data usage in distributed systems. IEEE Trans. Software Eng. **39**(10), 1403–1426 (2013)
13. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
14. Basin, D., Klaedtke, F., Zalinescu, 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)
15. 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)
16. 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)
17. Beame, P., Koutris, P., Suciu, D.: Skew in parallel query processing. In: R. Hull, M. Grohe (eds.) PODS 2014, pp. 212–223. ACM (2014)
18. Beame, P., Koutris, P., Suciu, D.: Communication steps for parallel query processing. J. ACM **64**(6), 40:1–40:58 (2017)
19. 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)
20. 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)
21. 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)
22. 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)
23. 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)
24. 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)
25. Cormode, G., Hadjieleftheriou, M.: Methods for finding frequent items in data streams. VLDB J. **19**(1), 3–20 (2010)
26. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI 2004, pp. 137–150. USENIX Association (2004)
27. Feng, S., Lohrey, M., Quaas, K.: Path checking for MTL and TPTL over data words. Log. Methods Comput. Sci. **13**(3) (2017)
28. Ganguly, S., Silberschatz, A., Tsur, S.: Parallel bottom-up processing of Datalog queries. J. Log. Program. **14**(1&2), 101–126 (1992)
29. Hallé, S., Khoury, R.: Event stream processing with BeepBeep 3. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, Kalpa Publications in Computing, pp. 81–88. EasyChair (2017)
30. 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)
31. Havelund, K., Peled, D.: Efficient runtime verification of first-order temporal properties. In: M. Gallardo, P. Merino (eds.) Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings, *Lecture Notes in Computer Science*, vol. 10869, pp. 26–47. Springer (2018)
32. 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)
33. 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)
34. 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)
35. 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)
36. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebr. Progr. **78**(5), 293–303 (2009)

37. 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
38. 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)
39. 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)
40. 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)
41. 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)
42. 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)
43. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Log. Methods Comput. Sci. **8**(1) (2012)
44. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Implementation associated with this paper. https://bitbucket.org/krle/scalable-online-monitor (2018)
45. 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)
46. 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)
47. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Isabelle formalization associated with this paper. https://bitbucket.org/krle/scalable-online-monitor/downloads/slicing-formalization.zip (2019)
48. 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)
49. 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)
50. 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)