Scalable Online First-Order Monitoring

Joshua Schneider, David Basin, Frederik Brix, Srđan Krstić, and Dmitriy Traytel

Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland

Abstract. Online monitoring is the task of identifying complex temporal patterns while incrementally processing streams of events. Existing state-of-the-art monitors can process streams of modest velocity in real-time: a few thousands events per second. We scale up monitoring to higher velocities by slicing the stream, based on the events' data values, into substreams that can be independently monitored. 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 the resulting automatic data slicer in Apache Flink and use the MonPoly tool to monitor the substreams. We empirically evaluate this setup, demonstrating a substantial scalability improvement.

1 Introduction

Large-scale software systems produce millions of log events per second. Identifying 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 event-wise, 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 [8, 13]. Propositional languages, however, are severely limited in their expressiveness. Since they regard events as atomic, they cannot formulate dependencies between data values stored in the events. First-order specification languages, such as metric first-order temporal logic (MFOTL) [11], do not have this limitation. Various online monitors [5,7,11,14,26,32,34] can handle first-order specification languages for event streams 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 different subpatterns in parallel. The amount of parallelization offered is limited by the number of subpatterns for a given pattern. The alternative is *data parallelism*: 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 paper 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 (i.e., velocity spikes).

Data parallelism has previously been utilized to scale up offline monitoring [9] (Section 2). Yet neither offline nor online monitoring is a data-parallel task in general. This means that, in some cases, parallel monitors must synchronize during their execution. Alternatively, careful data duplication across the parallel monitors allows for a non-blocking parallel architecture. An important contribution of this prior work on scalable offline monitoring is the development of a (*data*) *slicing framework* [9]. The framework takes as inputs an MFOTL formula (Section 3) and a splitting strategy that determines which parallel monitors the data should be sent to. It outputs an event dispatcher that forwards events to appropriate monitors and ensures that the overall parallel architecture collectively produces exactly the same results that a single-threaded monitor would do.

The previous slicing framework has three severe limitations. First, data can be sliced on only one free variable at a time. Although the single-variable slices can be composed into multi-variable slices, the composition does not offer the flexibility of simultaneously slicing on multiple variables. As a result, composition is ineffective for some formulas and adds unnecessary data duplication for others. Second, the user of the slicing framework must supply a slicing 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 paper addresses all of the above limitations with the following contributions:

- We generalize the offline slicing framework [9] 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 the hypercube algorithm [2,27] used previously to parallelize via hashing implementations of relational join operators in databases.
- We implement our new slicing framework using the Apache Flink [3] stream processing engine (Section 6). We use MonPoly [11, 12] as a black-box monitor for the slices. A particular challenge in our publicly available implementation [35] was to efficiently checkpoint MonPoly's state within Flink to achieve fault-tolerance.
- We evaluate the slicing framework and automatic strategy selection on both synthetic data and real-world data based on Nokia's data collection campaign [10] (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 on the formulas used in the Nokia case study.

2 Related work

Our work builds on the slicing framework introduced by Basin et al. [9]. 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 based on time. We have generalized their data slicers to operate simultaneously on all free variables in a formula. Moreover, the usage of MapReduce in the implementation of the original framework limited it to the offline setting. In contrast, our implementation in Apache Flink 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. [4], Bianculli et al. [18], and Bersani et al. [17] 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 [34] 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. [5] generalize this approach to more complex properties expressed using quantified event automata (QEA). Reger and Rydeheard [32], 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 [28] show that the LTL monitoring problem belongs to AC^1(logDCFL) and as such 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 [19] and Feng et al. [22] who study variants of MTL and TPTL.

Complex event processing (CEP) systems analyze streams by recognizing composite events as (temporal) patterns of simple events. Their publish-subscribe architecture allows for ample parallelism. The languages used by CEP systems are often based on SQL extensions without a clear semantics. An exception is BeepBeep [24]: a multi-threaded [25] 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 partitioning scheme (also known as the *shares algorithm*) was first proposed by Afrati and Ullman [2] in the context of MapReduce. The idea underlying the algorithm can be traced back to the parallel evaluation of datalog queries [23]. The hypercube algorithm was shown to be optimal for conjunctive queries with one communication round on skew-free databases [16].

The hypercube and other hash-based partitioning schemes are sensitive to skew. Rivetti et al. [33] suggest a greedy balancing strategy after separating the heavy hitters, i.e., frequently occurring input values. This approach is restricted to multi-way joins in which all relations share a common join key. Nasir et al. [29, 30] balance skew for associative stream operators without identifying heavy hitters explicitly. Vitorovic et al. [37] combines the hash-based hypercube, prone to heavy hitters, with random partitioning [31], resilient to heavy hitters. The 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 general MFOTL formulas. Instead we follow a hypercube variant, which is optimal for the considered setting with skew [27]. The heavy hitters must be known in advance in this approach.

$$
\begin{array}{ll|ll}
v, i \models r(t_1, \ldots, t_n) & \text{if } r(v(t_1), \ldots v(t_n)) \in D_i & v, i \models \exists x.\ \varphi & \text{if } v[x \mapsto z], i \models \varphi \text{ for some } z \in \mathbb{D} \\
v, i \models t_1 \approx t_2 & \text{if } v(t_1) = v(t_2) & v, i \models \bullet_I \varphi & \text{if } i > 0,\ \tau_i - \tau_{i-1} \in I, \text{ and } v, i-1 \models \varphi \\
v, i \models \neg\varphi & \text{if } v, i \not\models \varphi & v, i \models \bigcirc_I \varphi & \text{if } \tau_{i+1} - \tau_i \in I \text{ and } v, i+1 \models \varphi
\end{array}
$$

$$
\begin{array}{ll}
v, i \models \varphi \lor \psi & \text{if } v, i \models \varphi \text{ or } v, i \models \psi \\
v, i \models \varphi \,\mathsf{S}_I\, \psi & \text{if } v, j \models \psi \text{ for some } j \leq i,\ \tau_i - \tau_j \in I, \text{ and } v, k \models \varphi \text{ for all } k \text{ with } j < k \leq i \\
v, i \models \varphi \,\mathsf{U}_I\, \psi & \text{if } v, j \models \psi \text{ for some } j \geq i,\ \tau_j - \tau_i \in I, \text{ and } v, k \models \varphi \text{ for all } k \text{ with } i \leq k < j
\end{array}
$$

Fig. 1: Semantics of MFOTL

3 Preliminaries

We briefly recall the syntax and semantics of our specification language of choice, metric first-order temporal logic (MFOTL) [11], and describe the monitoring setting considered.

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 $[a, b) := \{x \in \mathbb{N} \mid a \leq x < b\}$, where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$ and $a < b$. Formulas φ are constructed inductively, where t_i, r, x, and I range over $\mathbb{V} \cup \mathbb{D}$, $\mathbb{E}$, $\mathbb{V}$, and $\mathbb{I}$, respectively:

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

Along with the Boolean operators, MFOTL includes the metric past and future temporal operators $\bullet$ (*previous*), S (*since*), $\bigcirc$ (*next*), and 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 $\varphi \land \psi := \neg(\neg\varphi \lor \neg\psi)$, 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) $\blacksquare_I \varphi := \neg \blacklozenge_I \neg\varphi$. Abusing notation, $\mathbb{V}_\varphi$ denotes the set of free variables of the formula φ.

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

The relation $v, i \models_\rho \varphi$ (Figure 1) defines the satisfaction of the formula φ for a valuation v at an index i with respect to the stream $\rho = \langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$. Whenever ρ is fixed and clear from the context, we omit the subscript on $\models$. The valuation v is a mapping $\mathbb{V} \to \mathbb{D}$, assigning domain elements to the free variables of φ. Overloading notation, v is also the extension of v to the domain $\mathbb{V} \cup \mathbb{D}$, 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 infinite streams, a monitor will always receive only a finite stream prefix. We write $\mathbb{P}$ for the set of finite stream prefixes and $\preceq$ for the usual prefix order on streams and stream prefixes. For the prefix π and $i \in \{1, \ldots, |\pi|\}$, $\pi[i]$ denotes π's i-th element.

Abstractly, a *monitor function* 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 on 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.

Concretely, the MonPoly monitor [12] implements a monitor function $\mathscr{M}_\varphi$ for φ from a practically relevant fragment of MFOTL [11]. Whenever φ is violated for a particular index, MonPoly outputs the valuations of the free variables that cause φ to become false. Because some violations may be found only after a time delay, we ignore the order of MonPoly's output and model it as a set, with $\sqsubseteq$ being the subset relation:

$$\mathscr{M}_\varphi(\pi) = \{(v,i) \mid i \leq |\pi| \text{ and for all } \rho \in \mathbb{S} \text{ with } \pi \preceq \rho,\ (v,i) \not\models_\rho \varphi\}.$$

4 Slicing Framework

We introduce a general framework for parallel online monitoring based on slicing. Basin et al. [9] 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 to monitor the slice. We adapt this idea to online monitoring. Since slicing with respect to time is not particularly useful in the online setting, we focus on the data in the events.

4.1 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 indicates how to split the stream prefix into independently monitorable substreams and how to combine the verdict outputs of the parallel submonitors into a single verdict.

Definition 1. *A* slicer *for a monitor function $\mathscr{M}$ 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 *$J \in (K \to \mathbb{O}) \to \mathbb{O}$ indicates how to combine K-indexed verdicts into one, satisfying the following properties:*

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 $k \in K$.*
Soundness *For all $\pi \in \mathbb{P}$, $J\big(\lambda k.\ M_k(S(\pi)_k)\big) \sqsubseteq \mathscr{M}(\pi)$.*
Completeness *For all $\pi \in \mathbb{P}$, $\mathscr{M}(\pi) \sqsubseteq J\big(\lambda k.\ M_k(S(\pi)_k)\big)$.*

For an input prefix π, $S(\pi)$ 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 $\mathscr{M}$, can process the sliced prefixes incrementally. Composing the splitter, the corresponding submonitor for each slice, and the joiner yields the parallelized monitor function $J\big(\lambda k.\ M_k(S(\pi)_k)\big)$. It is sound and complete if and only if it computes the same verdicts as $\mathscr{M}$.

For example, parametric trace slicing [32, 34] can be seen as a particular slicer for monitor functions that arise from sliceable FO-LTL formulas [32, Section 4]. Thereby, K is the cross-product of finite domains for the formulas' variables. Thus elements of K are 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. In practice, we would like a highly efficient implementation of S since it is a centralized operation in front of the parallel inner monitors M, which must inspect every input event. Parametric trace slicing determines the target slice for an event by inspecting events individually (and not as part of the whole prefix). We call splitter with this property event-separable. Event-separable splitters are desirable because they cater to a parallel implementation of the splitter S.

Definition 2. *A splitter S is called* event-separable *if there is a function $\hat{S} \in (\mathbb{E} \times \mathbb{D}^*) \to \mathscr{P}(K)$ such that $S(\pi)_k[i] = \langle \tau_i, \{e \in D_i \mid k \in \hat{S}(e)\}\rangle$ for all $\pi \in \mathbb{P}$, $k \in K$, $i \leq |\pi|$.*

Lemma 1. *If S is event-separable, then $\pi_1 \preceq \pi_2$ implies $S(\pi_1)_k \preceq S(\pi_2)_k$ for all $k \in K$.*

We also call slicers with event-separable splitters *event-separable*. We identify event-separable slicers (K, M, S, J) with $(K, M, \hat{S}, J)$.

4.2 Joint Data Slicer

We describe an event-separable slicer for the monitor function $\mathscr{M}_\varphi$ that arises from the MFOTL formula φ. Our *joint data slicer* distributes events according to the valuations they induce in the formula. Recall that the output of $\mathscr{M}_\varphi$ consists of all valuations that do not satisfy the formula at some timepoint. We would like to evaluate φ for each valuation to determine whether the valuation is a violation. 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.

We assume that the bound variables in φ are disjoint from the free variables. Given an event $e = r(d_1, \ldots, d_n)$, the set $matches(\varphi, e)$ contains all valuations $v \in \mathbb{V}_\varphi \to \mathbb{D}$ for which there is a subformula $r(t_1, \ldots, t_n)$ in φ where $v(t_i) = d_i$ for all $i \in \{1, \ldots, n\}$. We implicitly extend v to $\mathbb{V} \cup \mathbb{D}$, such that it is the identity on $(\mathbb{V} \setminus \mathbb{V}_\varphi) \cup \mathbb{D}$.

Definition 3. *Let φ be an MFOTL formula and $f \in (\mathbb{V}_\varphi \to \mathbb{D}) \to \mathscr{P}(K)$ be a mapping from valuations to nonempty sets of slice identifiers. The* joint data slicer *for φ with splitting strategy f is the tuple $(K, \lambda k.\ \mathscr{M}_\varphi, \hat{S}_f, J_f)$, where*

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

The intersection with $\{v \mid k \in f(v)\} \times \mathbb{N}$ in the definition of J_f is needed only for some formulas, notably those that involve equality. Consider, e.g., the formula $x \approx a \to Q(x)$, where a is a constant. Even if a prefix contains $Q(a)$, so that no violation occurs,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 verdict from those slices, the result will be unsound.

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

Proof. Monotonicity follows from Lemma 1. For soundness and completeness, we must show that $(v,i) \in J_f(\mathscr{M}_\varphi \circ S_f(\pi))$ if and only if $(v,i) \in \mathscr{M}_\varphi(\pi)$ for an arbitrary v and i. By the definitions of J_f and $\mathscr{M}_\varphi$, this is equivalent to

$$(\forall k \in f(v). \forall \rho.\ \rho \succeq S_f(\pi)_k \implies v,i \models_\rho \neg\varphi) \iff (\forall \sigma.\ \sigma \succeq \pi \implies v,i \models_\sigma \neg\varphi).$$

For an arbitrary $k \in f(v)$, it suffices to consider streams ρ and σ such that $\rho[i] = S_f(\pi)_k[i]$ and $\sigma[i] = \pi[i]$ for all $i \leq |S_f(\pi)_k| = |\pi|$, and $\rho[i] = \sigma[i]$ otherwise. Then $v,i \models_\rho \varphi \iff v,i \models_\sigma \varphi$ follows by induction on the structure of φ generalizing over v and i. □

Example 1. Consider the formula $P(x,y) \to \Diamond_{[0,5]}(P(y,x) \wedge Q(x))$. We apply the joint data slicer for two slices ($K = \{1,2\}$). Its splitting strategy f maps all valuations to the first slice, except for $\langle x = 5, y = 7\rangle$. (Note that the intersection with $\{v \mid k \in f(v)\} \times \mathbb{N}$ is redundant in the definition of J_f for this formula. For any violating valuation, there must be a matching P event in the slice, which determines the values of all free variables.)

Let π be the prefix $\langle 11, \{P(5,1), Q(2)\}\rangle, \langle 12, \{P(5,7), Q(3), Q(5)\}\rangle, \langle 21, \{P(7,5)\}\rangle$. We obtain the slices

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

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 data slicer used in the offline slicing framework [9] is defined for a single free variable x and a collection $(S_k)_{k \in K}$ of slicing sets. Each slicing set is a subset of the domain, and $\bigcup_{k \in K} S_k = \mathbb{D}$. 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 a composition of single variable slicers. Consider a formula with the atoms $P(x,y)$, $P(y,z)$, and $P(z,x)$. Any single variable slicer will send all P events to all slices because each atom misses one free variable. We show in Section 5 that our joint data slicer is more precise for such formulas.

5 Automatic Slicing

The joint data slicer (Section 4.2) is parameterized by a splitting strategy. Ideally, the chosen strategy optimally utilizes the available computing resources. In particular, computation and communication costs should be evenly distributed, while keeping the overhead low. As an approximation of the overall computational cost, we consider the event rate, i.e., the number of events in a period of time, of each slice. We chose this as minimizing the event rate should reduce the submonitors' execution time and memory consumption. We do not optimize the amount of communication in this paper. However, the number of slices is a parameter that affects the communication cost due to data duplication.

We base our splitting strategy on the hypercube algorithm [2,16,23,27,36]. Below, we describe this algorithm within our framework and address challenges that arise in the online setting. We decided on the hypercube algorithm as our starting point because its skew-aware variant by Koutris et al. [27] has been shown to yield strategies that are optimal

with respect to the worst-case load for conjunctive queries. Query evaluation and monitoring are closely related: reporting violations of some formula φ with free variables is equivalent to evaluating the query $\neg\varphi$. Therefore, conjunctive queries constitute a specific subset of all monitoring tasks. While this does not imply optimality for arbitrary formulas, we are still able to effectively slice formulas containing operators other than conjunctions.

Let $[n] = \{1,\ldots,n\}$ for $n \in \mathbb{N}$. We assume a linear ordering $x_1,\ldots,x_k$ on the free variables of the formula φ. The basic idea of the hypercube algorithm is to organize N submonitors into a hypercube (or more precisely, an orthotope) $K = [n_1] \times \ldots \times [n_k]$ such that $\prod_{i\in[k]} n_i = N$. The parameters $n_1,\ldots,n_k$ are called *shares*. At the beginning of the monitoring, hash functions $h_i \in \mathbb{D} \to [n_i]$ are chosen randomly and independently. To decide which submonitors should receive an event, the hypercube algorithm applies the hash function h_i to the values of the free variable x_i, for all $x_i \in \mathbb{V}_\varphi$. This is done for each valuation that is inferred from the event by the data slicer. Each tuple of hash values represents a coordinate in K, which is the target slice for that valuation. Therefore, the splitting strategy f maps valuations v to (sets of) slice identifiers $\{\langle h_1(v(x_1)),\ldots,h_k(v(x_k))\rangle\}$.

The shares are chosen to optimize the maximum event rate over all submonitors. For now, we assume that all values have low *degree*. The degree of a data value denotes the number of events that contain this value in a specific attribute. Then, the maximum event rate of the monitors given the shares n is estimated by the following cost function [2,15]:

$$cost(n,\varphi,Z) = \sum\nolimits_{r(d_1,\ldots,d_n)\in\varphi} \frac{Z(r)}{\prod_{x_i\in\{d_1,\ldots,d_n\}\cap\mathbb{V}_\varphi} n_i}.$$

The term $r(d_1,\ldots,d_n)$ ranges over all atoms in the formula φ. The function Z is a parameter of the optimization, where $Z(r)$ is the rate of events with name r. We use a simplified version of the algorithm by Chu et al. [21] to optimize the cost function. The algorithm enumerates all possible integer shares with product N. This is feasible because the number of share combinations is small for realistic N, even when N has many small prime divisors. In fact, we assume that N is a power of two in our implementation.

It is not obvious how the statistics Z can be meaningfully obtained in the context of online monitoring. We cannot collect them from the entire stream for two reasons. First, it is impossible to observe future events at the time when the splitting strategy must be fixed. Second, if the rates increase due to a temporary change in the stream characteristics, some monitor instance might become overwhelmed, events must be buffered, and latency increases. The optimization cannot account for such local changes if we use statistics collected over a long period of time, as one would do in offline processing. However, some buffering to handle transient rate spikes is acceptable. Therefore, we estimate the statistics using a recorded prefix of the stream and aggregate them over short time windows.

Example 2. Consider the formula $\varphi = P(x,y) \wedge Q(y,z) \rightarrow \neg R(z,x)$ and a stream consisting of $3m$ events in a given period of time. We assume that the events with names P, Q, and R occur equally often, such that $Z(P) = Z(Q) = Z(R) = m$. The optimal shares for the hypercube composed of N monitors are $n_x = n_y = n_z = N^{1/3}$. Each slice contains approximately $cost(n,\varphi,Z) = 3m/N^{2/3}$ events. We obtain the same results for the formula $\varphi' = P(x,y) \wedge P(y,z) \rightarrow \neg P(z,x)$ and $Z(P) = m$ because each event is replicated three times. Yet, the event rate per slice is lower than the rate of the input stream if $N \geq 8$. This is an improvement over the single variable slicer (Section 4.2).

Next, we show how the joint data slicer with the optimal hypercube strategy for φ distributes some events. We assume $N = 64$ and simplify the hash functions to $h(x) = x \bmod 4$. The slices are thus identified by strings of three numbers between 0 and 3, with one number for each variable x, y, and z.

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

The events $P(0,1)$, $Q(1,7)$, and $R(7,0)$ are sent to slice 013, which ensures completeness.

Data distributions without skew, in which all values have low degree, are too limited in practice. Hash-based partitioning schemes such as the hypercube fail to distribute events evenly if the input is skewed. A standard solution it to use a different splitting strategy for the highly skewed portions of the data [27]. Following the terminology of [27], a *heavy hitter* is a value with degree at least $Z(r)/N$ in events with name r. We collect heavy hitter information along with the Z statistics. A value is considered a heavy hitter if it is a heavy hitter in at least one of the time windows over which we collect the Z statistics.

To compute the slice for a valuation, our skew-aware strategy first determines the set H of variables that are heavy hitters. A value d is a heavy hitter for a variable x if there is an atom $r(\dots,t_{j-1},x,t_{j+1},\dots)$ in the formula such that d is a heavy hitter in the j-th attribute of r. For each set H, a separate collection of shares n_x^H and independent hash functions h_x^H is used, where $x \in \mathbb{V}_\varphi$. Note that there are 2^k different sets H, where $k = |\mathbb{V}_\varphi|$. For $x \in H$, we fix $n_x^H = 1$ in the share optimization. The remaining shares are computed as before, as is $f(v)$, but using the hash functions h_x^H.

Example 3. Consider the same formula φ and stream as before, but suppose now that the stream has some heavy hitters. We analyze the optimal shares for the heavy-hitter sets $A = \{x\}$, $B = \{x,y\}$, and $C = \{x,y,z\}$. The remaining cases have symmetrical solutions. In the case A, the shares are $n_x^A = 1$ and $n_y^A = n_z^A = N^{1/2}$. Each slice then contains at most $1/N^{1/2}$ of the events for which only x is assigned a heavy hitter. In the case B, the optimal shares are $n_x^B = n_y^B = 1$ and $n_z^B = N$, so there are at most $1/N$ of the corresponding events in each slice. Finally in the case C, one must broadcast the events. Note that there can be at most N different heavy hitters per attribute. Therefore, there are at most $3wN^2$ events to which the set C applies, where w is the number of databases in the time period that we consider. If w is bounded by a constant, the overall fraction of events in each slice is asymptotically equal to the maximum of the three cases, which is $O(1/N^{1/2})$.

Now assume that 0 is a heavy hitter in the first attribute of P, and $N = 64$. Therefore, we need to consider the heavy-hitter sets $\{\}$ and $\{x\}$. Let the hash functions be the modulus as in the previous example (e.g., $h_y^{\{x\}}(y) = y \bmod 8$). We overlay the slices for the different heavy-hitter sets and assign identifiers from $[N]$ according to their lexicographic ordering: both valuations $\langle x=1, y=1, z=3\rangle$ and $\langle y=2, z=7\rangle$ map to 23.

event	containing slices	event	containing slices
$P(0,1)$	$8,9,10,11,12,13,14,15$	$Q(1,7)$	$7,23,39,55;15$
$P(1,1)$	$20,21,22,23$	$R(7,0)$	$7,15,23,31,39,47,55,63$

Both $\{\}$ and $\{x\}$ are possible for $Q(1,7)$ because $Q(y,z)$ does not induce a valuation for x.

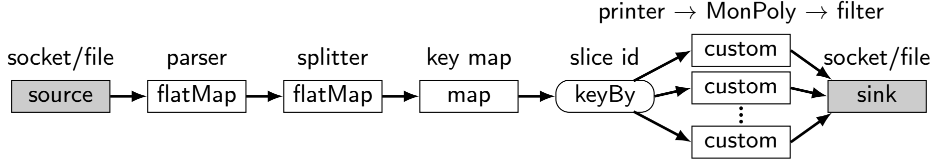


Fig. 2: Stream operators in the online monitor

In general, the possible rate reduction depends on the pattern of free variables in the formula's atoms. A detailed discussion is provided by Koutris et al. [27]. 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 proportional to $1/N$. Atoms with missing variables, and equivalently variables with heavy hitters, increase the fraction to $1/N^q$ for some $q > 1$. 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 [9].

6 Implementation

We implemented a parallel online monitor based on the joint data slicer in Scala on top of the Apache Flink stream processing framework. Given a formula, the slicer reads events from a TCP socket or a text file, monitors them in parallel, and writes the collected verdicts to a second socket or file. The monitoring of the slices is delegated to MonPoly [12].

The Flink API provides the means to construct a logical dataflow graph. The graph consists of operators that retrieve data streams from external sources, apply processing functions to stream elements, and output the elements to sinks. Operators and the flows between them can execute in parallel; elements are partitioned according to user-specified keys. At runtime, Flink maps the graph to a distributed cluster of computing nodes. We chose Flink for its low latency stream processing and its support for fault tolerant computation. Fault tolerance is ensured using a distributed checkpointing mechanism [20]: The system recovers from failures by restarting from regularly created checkpoints. Operators must expose their state to the framework to participate in the checkpoints.

Figure 2 shows the dataflow graph of our slicer. Its main parameters are the number N of parallel monitors and the inputs for the shares optimization, which is performed during initialization. Events are read line by line as strings. We support both MonPoly's input format and the CSV format used in the first RV competition [6]. The parser converts the input lines into an internal datatype that stores the event name and the list of data values. We flatten the parser's results into a stream of single events because a single line in MonPoly's format may describe several events, i.e., an entire database, at once.

After parsing, the splitter computes the set of target slices for each event. To do so, it first determines the matching valuations as described in Section 4.2. As this set may be infinite, the splitter encodes it as a partial mappings from variables to data values. For each target slice, a copy of the event is sent to the next operator along with the slice identifier.

At this point, we would like to distribute the events to N parallel submonitors. However, the keyBy operation in the Flink API applies its own hash function to shuffle the events. This hash function might needlessly collapse slice identifiers. We work around this limitation by mapping the identifiers to preimages under the hash function, which we precompute by enumeration. In each parallel flow, a custom operator prints the internal datatype in MonPoly format, sends it to an external monitor process, and applies the intersection from the definition of J_f (Definition 3) to its output. Finally, all remaining verdicts are combined into a single stream, which is written to an output socket or file.

The custom operator is responsible for starting and interacting with the MonPoly process. The operator writes one database at a time to standard input and simultaneously reads violations from standard output of the process. Reading and writing is asynchronous to the Flink pipeline in order to prevent blocking other operators. Flink's `AsyncWaitOperator` supports asynchronous requests to external processes without managing their state. We must, however, include the submonitors' states in the checkpointing because they summarize the events seen so far. Thus, our custom operator tracks MonPoly's state. To this end, we extend MonPoly with control commands for saving and loading its state. Whenever Flink instructs the custom operator to create a checkpoint, it first waits until all prior events have been processed. Then, the command for saving state is issued and MonPoly writes its state to a temporary file. Violations reported after the checkpoint instruction's arrival at the custom operator are included in the checkpoint. This ensures that no violation is lost because other operators might create their own checkpoint concurrently. We evaluate the overhead of checkpointing in Section 7.

The part of the dataflow before the submonitors is not parallel. This is a bottleneck that limits scalability: all events in the input must be processed sequentially by the splitter. Despite this limitation of our implementation, the splitter and the surrounding operators could be parallelized too: As its splitter operates on individual events, the joint data slicer is event-separable (Section 4.1). A parallel splitter would be particularly effective if the event source itself is distributed. However, we would have to ensure that events arrive at the submonitors in chronological order after reshuffling. This order is no longer guaranteed if the splitter is partitioned into concurrent tasks. A possible solution is to buffer and reorder events. We leave the analysis of such an extension to future work.

7 Evaluation

With our evaluation, we aim to answer the following research questions:

RQ1: How does our monitor scale with respect to the index and event rate?
RQ2: How does our monitor scale with respect to different variable occurrence patterns?
RQ3: How much overhead is incurred by supporting fault tolerance (FT)?
RQ4: Can knowledge about the frequency of event names improve performance?
RQ5: Can knowledge about heavy hitter values improve performance?
RQ6: Can our monitor handle data in a real-world online setting?
RQ7: How does it compare to MonPoly running on a single core?

To answer the research questions, we organize our evaluation into two families of experiments, each monitoring a different type of input stream (synthetic or real-world). The synthetic streams are used to analyze the effects of individual parameters, e.g., event rate, while real-world streams attest to our tool's ability to scalably solve realistic problems.

$$
\begin{aligned}
\mathit{star} &= \Box \forall a.\forall b.\forall c.\forall d.\ (\blacklozenge_{[0,10s)} P(a,b)) \rightarrow Q(a,c) \rightarrow \Box_{[0,10s)} \neg R(a,d) \\
\mathit{linear} &= \Box \forall a.\forall b.\forall c.\forall d.\ (\blacklozenge_{[0,10s)} P(a,b)) \rightarrow Q(b,c) \rightarrow \Box_{[0,10s)} \neg R(c,d) \\
\mathit{triangle} &= \Box \forall a.\forall b.\forall c.\quad (\blacklozenge_{[0,10s)} P(a,b)) \rightarrow Q(b,c) \rightarrow \Box_{[0,10s)} \neg R(c,a)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{script} &= \Box \forall db.\forall dt.\mathit{select}(\mathsf{script1},db,dt) \vee \mathit{insert}(\mathsf{script1},db,dt) \vee \\
&\qquad \mathit{delete}(\mathsf{script1},db,dt) \vee \mathit{update}(\mathsf{script1},db,dt) \rightarrow \\
&\big((\neg \blacklozenge_{[0,1s)} \lozenge_{[0,1s)} \mathit{end}(\mathsf{script1}))\ \mathsf{S}\ (\blacklozenge_{[0,1s)} \lozenge_{[0,1s)} \mathit{start}(\mathsf{script1}))\big) \vee \blacklozenge_{[0,1s)} \lozenge_{[0,1s)} \mathit{end}(\mathsf{script1}) \\
\mathit{insert} &= \Box \forall u.\forall dt.\mathit{insert}(u,\mathsf{db1},dt) \wedge dt \not\approx \mathsf{unknown} \rightarrow \\
&\quad \blacklozenge_{[0,1s)} \lozenge_{[0,30h]} \exists u'.\mathit{insert}(u',\mathsf{db2},dt) \vee \mathit{delete}(u',\mathsf{db1},dt) \\
\mathit{delete} &= \Box \forall u.\forall dt.\mathit{delete}(u,\mathsf{db1},dt) \wedge dt \not\approx \mathsf{unknown} \rightarrow \\
&\quad \big(\blacklozenge_{[0,1s)} \lozenge_{[0,30h)} \exists u'.\mathit{delete}(u',\mathsf{db2},dt)\big) \vee \\
&\quad \big((\lozenge_{[0,1s)} \blacklozenge_{[0,30h)} \exists u'.\mathit{insert}(u',\mathsf{db1},dt)) \wedge (\blacksquare_{[0,30h)} \Box_{[0,30h)} \neg \exists u'.\mathit{insert}(u',\mathsf{db2},dt))\big)
\end{aligned}
$$

Fig. 3: MFOTL formulas used in the evaluation

We implemented a generator that takes a random seed and synthesizes streams with specific characteristics. It produces streams containing binary events labeled with P, Q, or R with configurable event rate, index rate, and rate of violations for the three fixed formulas *star*, *linear*, and *triangle* (Figure 3). This setup allows us to test RQ1. Furthermore, to test RQ4 and RQ5, the generator synthesizes event labels with configurable rates ($Z(P)$, $Z(Q)$, and $Z(R)$) and forces some event attribute values to be heavy hitters.

We use logs collected during the Nokia's Data Collection Campaign [10] as real-world streams. The campaign collected data from mobile phones of 180 participants and propagated it 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, database triggers on db2 anonymized and copied the data to db3. The participants could query and delete their own data in db1 and such 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 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 from the log with the highest average event rate and we use our replayer tool to speed it up between 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 speedup, we have subjected our tool to streams of over half a million events per second. The used logs [1] and the scripts that synthesize and replay streams [35] are publicly available.

Figure 3 shows the formulas we monitored in our evaluation. The formulas *star*, *linear*, and *triangle* are tailored for the synthetic streams. Different occurrence patterns of free variables in the formulas allows us to test RQ2. We aimed to cover common data patterns in database queries [16] and extend them with temporal aspects. The formulas *script*, *insert*, and *delete* stem from the Nokia's Data Collection Campaign and have been shown there to be challenging to monitor [10]. Since we monitor only a one day fragment of the log from Nokia, 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, store the monitor's state, and start the monitor with the stored state as its initial state in the experiments.

We ran all our experiments on a server with two sockets, each containing twelve Intel Xeon 2.20GHz CPU cores with hyperthreading that effectively gives us 48 independent computation threads. 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 until the moment the monitor processes the last emitted event. We also measure the maximal memory usage of each submonitor. To measure latency during execution, the replayer tool injects a special event (called a latency marker) into the stream tagged with the current time. The marker is propagated by the monitor and the latency is measured at the output by comparing the current time with the time in the marker's tag. Besides the current latency measurement, we also calculate the rolling average and maximum latency up to the current point in the experiment. Flink supports latency markers and provides us with separate latency measurements for each operator in our monitor's implementation. Our replayer tool generates latency markers every second. 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. 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.

Figure 4 shows the results for synthetic streams. Figure 4a (top) shows the maximum latency when monitoring the formula *star* with different numbers of cores. With the event rate of 2,200 events per second, the single core variant of our tool already exhibits 5 second latency. Similar latency is exhibited with 4 cores when monitoring events rates above 8,000. In contrast, using 16 cores achieves sub-second latency for all event rates in our experiments. We also show the maximal memory consumption across all submonitors in Figure 4a (bottom). With an increasing number of submonitors, each submonitor receives fewer events and hence uses less memory. This experiment answers RQ1: our tool handles significantly higher event rates by using more parallel submonitors.

Figure 4b shows maximum latency (top) and memory consumption (bottom) of our tool when monitoring *star*, *triangle*, and *linear* formulas using 4 cores. 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 1 or 1,000. Since the index rate affects the performance of MonPoly [11], our tool is also affected. The event rate gain enabled by more submonitors depends on the variable occurrence patterns in the monitored formula (RQ2). Figure 4b also demonstrates that the *star* pattern is the one that exhibits the best scalability.

In the above experiments, we did not supply our monitor with the information on the rates of event labels in the stream. Figure 4c positively answers RQ4 by showing that both our tool's latency (top) and memory consumption (bottom) decrease independently of the number of cores when such statistics about the stream are known in advance.

Figure 5 summarizes the results of monitoring the real-world log from the Nokia case study. The event and index rates are defined by the log, while we only control the replay speed. The experiments answer RQ7 and show that we achieve better performance

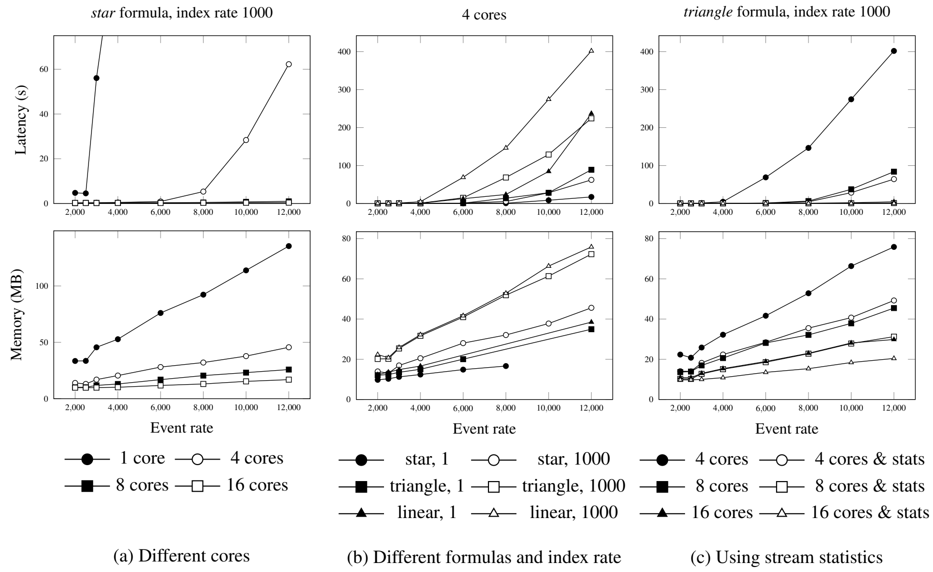


Fig. 4: Monitoring synthetic streams with fault tolerance

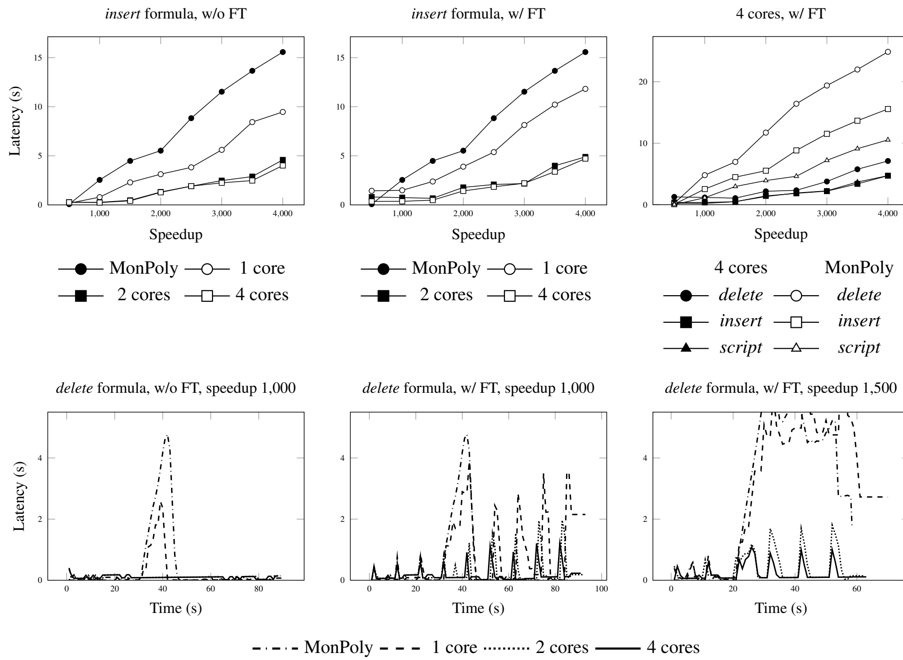
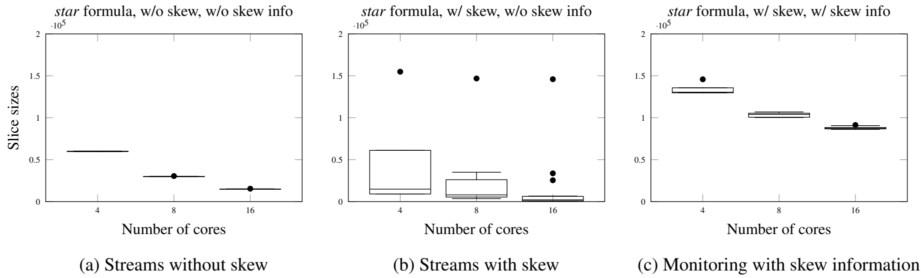


Fig. 5: Monitoring the real-world stream



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

Fig. 6: Impact of the skew and skew information on parallel monitoring

then MonPoly on its own, on all three formulas. We improve latency even when using a single core, due the optimized implementation of the slicer that filters the unnecessary events more efficiently than MonPoly. In this experiment, our tool's performance does not improve beyond 4 cores, since for event rates higher than 500,000 the centralized parsing and slicing becomes a bottleneck. The top left and middle plots contrast the performance overhead of fault tolerance (RQ3). The maximal latency is not affected; however the bottom three plots show that the current and average latency are. These plots correspond to three individual runs and depict how the current latency changes over time. The leftmost plot shows the monitoring of the *delete* formula with respect to the stream sped up 1,000 times, not accounting for fault tolerance. The middle and rightmost plots show runs with enabled fault tolerance support for the speedups of 1,000 and 1,500. The regular spikes in the current latency stem from Flink's state snapshot algorithm.

Figure 6a 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 6b). 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 6c), effectively reducing the maximum load of the submontors.

8 Conclusion and Future Work

We generalized the offline slicing framework [9] to support online monitoring and the simultaneous slicing with respect to all free variables in the formula. We adapted hash-based partitioning techniques from databases to obtain an automatic slicing strategy. We implemented the automatic slicing combining MonPoly with Flink and experimentally demonstrated a significant performance improvement: while retaining sub-second latency, 16-way parallelization allows us to increase the event rate by one order of magnitude.

We plan to extend our framework to include slicing on bound variables and to optimize slicing of rigid predicates. Checkpointing MonPoly's state coupled with the online collection of the stream's varying statistics can be used to dynamically reconfigure the automatic slicing strategy. We intend to implement this natural extension and analyze the tradeoff between the reconfiguration costs and the cost of using imperfect statistics. We also plan to refine our automatic splitting strategy to take communication costs explicitly into account and evaluate our approach on a distributed computing cluster.

Acknowledgment. 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).

References

1. The Nokia case study log file. `https://sourceforge.net/projects/monpoly/files/ldcc.tar/download` (2014)
2. Afrati, F.N., Ullman, J.D.: Optimizing multiway joins in a map-reduce environment. IEEE Trans. Knowl. Data Eng. 23(9), 1282–1298 (2011)
3. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The Stratosphere platform for big data analytics. VLDB J. 23(6), 939–964 (2014)
4. Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.A., Hallé, S.: MapReduce for parallel trace validation of LTL properties. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 184–198. Springer (2012)
5. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer (2012)
6. Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 1–9. Springer (2014)
7. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification, LNCS, vol. 10457, pp. 1–33. Springer (2018)
8. Basin, D., Bhatt, B., Traytel, D.: Almost event-rate independent monitoring of metric temporal logic. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 94–112. Springer (2017)
9. 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)
10. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: Monitoring data usage in distributed systems. IEEE Trans. Software Eng. 39(10), 1403–1426 (2013)
11. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM 62(2), 15:1–15:45 (2015)
12. Basin, D., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017)
13. Basin, D., Krstić, S., Traytel, D.: Almost event-rate independent monitoring of metric dynamic logic. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 85–102. Springer (2017)
14. Bauer, A., Küster, J., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer (2013)
15. Beame, P., Koutris, P., Suciu, D.: Skew in parallel query processing. In: Hull, R., Grohe, M. (eds.) PODS 2014. pp. 212–223. ACM (2014)
16. Beame, P., Koutris, P., Suciu, D.: Communication steps for parallel query processing. J. ACM 64(6), 40:1–40:58 (2017)
17. Bersani, M.M., Bianculli, D., Ghezzi, C., Krstić, S., Pietro, P.S.: Efficient large-scale trace checking using MapReduce. In: Dillon, L.K., Visser, W., Williams, L. (eds.) ICSE 2016. pp. 888–898. ACM (2016)

18. Bianculli, D., Ghezzi, C., Krstić, S.: Trace checking of metric temporal logic with aggregating modalities using MapReduce. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 144–158. Springer (2014)
19. Bundala, D., Ouaknine, J.: On the complexity of temporal-logic path checking. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8573, pp. 86–97. Springer (2014)
20. 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)
21. Chu, S., Balazinska, M., Suciu, D.: From theory to practice: Efficient join query evaluation in a parallel database system. In: Sellis, T.K., Davidson, S.B., Ives, Z.G. (eds.) SIGMOD 2015. pp. 63–78. ACM (2015)
22. Feng, S., Lohrey, M., Quaas, K.: Path checking for MTL and TPTL over data words. Log. Methods Comput. Sci. 13(3) (2017)
23. Ganguly, S., Silberschatz, A., Tsur, S.: Parallel bottom-up processing of Datalog queries. J. Log. Program. 14(1&2), 101–126 (1992)
24. Hallé, S., Khoury, R.: Event stream processing with BeepBeep 3. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. pp. 81–88. Kalpa Publications in Computing, EasyChair (2017)
25. Hallé, S., Khoury, R., Gaboury, S.: Event stream processing with multiple threads. In: Lahiri, S.K., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 359–369. Springer (2017)
26. Havelund, K., Peled, D., Ulus, D.: First order temporal logic monitoring with BDDs. In: Stewart, D., Weissenbacher, G. (eds.) FMCAD 2017. pp. 116–123. IEEE (2017)
27. Koutris, P., Beame, P., Suciu, D.: Worst-case optimal algorithms for parallel query processing. In: Martens, W., Zeume, T. (eds.) ICDT 2016. LIPIcs, vol. 48, pp. 8:1–8:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2016)
28. Kuhtz, L., Finkbeiner, B.: LTL path checking is efficiently parallelizable. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S.E., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 235–246. Springer (2009)
29. 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: Gehrke, J., Lehner, W., Shim, K., Cha, S.K., Lohman, G.M. (eds.) ICDE 2015. pp. 137–148. IEEE Computer Society (2015)
30. 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)
31. Okcan, A., Riedewald, M.: Processing theta-joins using mapreduce. In: Sellis, T.K., Miller, R.J., Kementsietsidis, A., Velegrakis, Y. (eds.) SIGMOD 2011. pp. 949–960. ACM (2011)
32. Reger, G., Rydeheard, D.E.: From first-order temporal logic to parametric trace slicing. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 216–232. Springer (2015)
33. Rivetti, N., Querzoni, L., Anceaume, E., Busnel, Y., Sericola, B.: Efficient key grouping for near-optimal load balancing in stream processing systems. In: Eliassen, F., Vitenberg, R. (eds.) DEBS 2015. pp. 80–91. ACM (2015)
34. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Log. Methods Comput. Sci. 8(1) (2012)
35. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Implementation associated with this paper. `https://bitbucket.org/krle/scalable-online-monitor` (2018)
36. Suri, S., Vassilvitskii, S.: Counting triangles and the curse of the last reducer. In: Srinivasan, S., Ramamritham, K., Kumar, A., Ravindra, M.P., Bertino, E., Kumar, R. (eds.) WWW 2011. pp. 607–614. ACM (2011)
37. 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)