Jumping the ORDER BY Barrier in Large-Scale Pattern Matching

Daniel Lupei
EPFL
daniel.lupei@epfl.ch

Mike Barnett
MSR, Redmond
mbarnett@microsoft.com

Saeed Maleki
MSR, Redmond
saemal@microsoft.com

Madan Musuvathi
MSR, Redmond
madanm@microsoft.com

Todd Mytkowicz
MSR, Redmond
toddm@microsoft.com

ABSTRACT

Event-series pattern matching is a major component of large-scale data analytics pipelines enabling a wide range of system diagnostics tasks. A precursor to pattern matching is an expensive "shuffle the world" stage wherein data are ordered by time and shuffled across the network. Because many existing systems treat the pattern matching engine as a black box, they are unable to optimizing the entire data analytics pipeline, and in particular, this costly shuffle.

This paper demonstrates how to optimize such queries. We first translate an expressive class of regular-expression like patterns to relational queries such that they can benefit from decades of progress in relational optimizers, and then we introduce the technique of *abstract pattern matching*, a linear time preprocessing step which, adapting ideas from symbolic execution and abstract interpretation, discards events from the input guaranteed not to appear in successful matches. Abstract pattern matching first computes a conservative representation of the output-relevant domain of every transition in a pattern based on the (unary) predicates of that transition. It then further refines these domains based on the structure of the pattern (i.e., paths through the pattern) as well as any of the pattern's join predicates across transitions. The outcome is an *abstract filter* that when applied to the original stream excludes events that are guaranteed not to participate in a match.

We implemented and applied *abstract pattern matching* in COSMOS/Scope to an industrial benchmark where we obtained up to 3 orders of magnitude reduction in shuffled data and 1.23x average speedup in total processing time.

1. INTRODUCTION

Event-series pattern matching has become essential to many data processing tasks as it enables complex behavioral, anomaly, and causality analyses, in varied domains ranging from network diagnostics and security breach detection, to algorithmic trading or click-path optimization. This trend prompted the addition of pattern matching constructs to many batch and stream processing engines such as Esper's Event Processing Language (EPL) [2], Oracle's `MATCH_RECOGNIZE` [4] or TerraData's `nPath` operator [1].

These languages let programmers specify patterns as a series of transitions, wherein a transition is triggered if the current event satisfies a guard defined in terms of both unary (selection) predicates as well as join conditions on priorly matched events. For example, a programmer might mine influential reviews within the click-stream of an e-commerce website consisting of events of type "Search"(S), "Read review"(R) and "Purchase"(P) by defining the pattern SR*P where each transition is joined on a userid (i.e., to make sure that all events in a match are correlated by the id of the user that performed them).

Pattern matching is usually only one of many stages in a data processing pipeline. As most of these stages are defined using relational queries (for eg., to enrich ingested data), the presence of pattern matching operators raises considerable challenges in terms of deriving optimum execution plans. Pattern matching operators do not enjoy the same wealth of rewriting rules and optimization opportunities as traditional relational operators and in addition require that their input is ordered by time. As a consequence, just sorting the data often takes a significant amount of processing resources even if matching the patterns themselves is relatively fast.

Warehoused data is typically processed by a mix of workloads which might comprise of both pattern mining and non-temporal queries (i.e., in which city are located the most active visitors of a website). These non-temporal queries have very different optimum data layout and since they are usually much more frequent, their optimum data layout ends up becoming the layout of choice in the data center. Keeping a second copy of the data sorted on time is wasteful when dealing with terabytes of data, especially when considering that many of the recorded events may not even be of interest to the mined pattern. We also note that the input data is not necessarily ordered by time to begin with, as it may be collected from a wide array of sources, each with varying constraints for when the data ingestion should happen. Finally, we remark that the requirement that the input be ordered by time is especially taxing when executing on a map-reduce platform, as the sorting step incurs an expensive reshuffling of the entire data.

In this work we demonstrate how to optimize a class of temporal queries on non-sorted data, thus reducing the cost of `ORDER BY time`. In particular, this paper introduces *abstract pattern matching*, a technique that builds cheap and

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

effective filters that remove a significant amount of data *before* sorting the data by time. Furthermore, our filters are themselves represented in relational algebra so the optimizer can include them in its optimization of the entire pipeline.

To gain an intuition for our approach, consider the earlier pattern SR*P for mining influential reviews. Abstract pattern matching first builds three independent sets of user ids: a set of user ids for users that (S)earched for a product, a set of user ids for users that (R)ead at least one review, and a set of user ids for users that (P)urchased the product. The intersection of these sets is a sound and conservative over-approximation of the set of users that will ultimately take part in the final match and thus those user ids not in this intersection can be filtered from the input (i.e., it is an over-approximation because it ignores time). This work formalizes and generalizes this intuition to more complex patterns that deal with multiple join (theta) predicates.

Abstract pattern matching first associates to every transition in a pattern a *symbolic set* capturing the domain of its join attributes based on those input events that satisfy its selection predicates. It then refines these symbolic sets by enforcing the join predicates between different transitions as well as the structure of the pattern. Finally, it selects from the input only those events that satisfy the resulting set of constraints, which we refer to as the *abstract filter*.

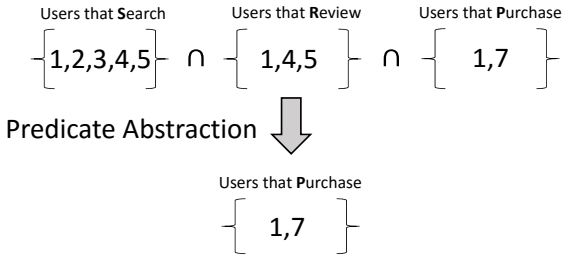
Since the precise representation of the symbolic sets could in many cases be just as large as the input, we introduce *data and predicate abstractions* to compute and query them in a time and space efficient manner. Depending on the type of join constraints that we have to propagate for a particular transition, *data abstraction* makes use of appropriate abstract set representations that can conservatively approximate those constraints (for example, for equijoins we make use of Bloom filters [8], whereas for inequality/band joins we rely on interval maps, as in Figure 1a). In addition, predicate abstraction further reduces the overheads of our approach by dropping in a sound way some of the transitions specified by the pattern. For example, few users ultimately purchase a product and so we can soundly over-approximate the set of users that may take part in an influential review by *only* computing that set (i.e., as in Figure 1b).

In concert, data and predicate abstraction allow us to cope with join predicates that do not have efficient data abstractions, as well as fine-tune our solution such that it only considers the most selective join predicates and balance the overhead of building our filters with their selectivity.

Our approach is inspired by the concept of abstract interpretation [16, 18]. In particular, the *abstract filter* constraints are the result of relaxing/coarsening in a conservative manner of the precise constraints enforced by the pattern regarding which events form successful matches. This suggests future work can leverage the technique of abstract interpretation to optimize other user defined aggregates as well, (i.e. to derive abstract filters meant to remove from the input those tuples that are guaranteed not to contribute to an output of interest).

As previously mentioned, constructing and applying the abstract filter incurs a series of overheads, most notably it requires a second pass over the data. Nonetheless, if the reduction in data is significant, these additional costs are balanced out by the dramatic speedup in the sorting/shuffling/pattern matching phases which results in an overall decrease in both processing costs and latency. Our experimental evaluation shows up to 3 orders of magnitude reduction in shuffled data

(a) Intervals are a compact over-approximation of the timestamps of searches.



(b) SR*P will only match those users that search, possibly read some reviews, and finally, purchase a product. Predicate abstraction exploits the selectivity of one predicate (most users do not purchase) to efficiently over-approximate users that match SR*P.

Figure 1: Examples illustrating the core intuitions underlying data and predicate abstraction.

as well as 1.23x average speedup in total processing time for 2 workloads: i) telemetry analysis over the events produced by an event-reporting infrastructure, and ii) repository analysis over the dataset of events published by GitHub. The reduction in total processing time is especially important in multi-tenant clusters where clients are billed based on the amount of computational resources they consume.

The contributions of this paper are:

- We show how a significant class of complex event patterns can be translated to relational queries such that they can benefit from decades of progress in relational optimizations.
- We introduce the technique of *abstract pattern matching* in order to minimize the sorting/shuffling costs of large scale mining of patterns within map-reduce frameworks.
- We design *data and predicate abstractions* that allow us to trade the precision of our approach (but not its soundness) for lower overheads.
- We prototype our solution and show on an industrial benchmark that it delivers significant reductions in the amount of data sorted/shuffled as well as processing times.

The rest of the paper is organized as follows: we further illustrate our approach and detail its design in sections 2 and 4, the related work is discussed in section 3, the choices we made in implementing our solution are explored in section 5, followed by the presentation of the results of our experimental evaluation in section 6. Finally, section 7 gives our concluding remarks and comments on future directions.

2. MOTIVATING EXAMPLE

Consider an analytics task that mines influential reviews within the click-stream of an e-commerce website consisting of events of type "Search"(S), "Read review"(R) and "Purchase"(P). The mining task is informally described in terms of the pattern SR*P. In other words, we desire a user interation where the user searched for an item, read a sequence of reviews, before deciding to purchase. In particular, we want the user to perform no other actions between the search and purchase actions.

The mining task is described in terms of the pattern SR*P, with each event variable annotated by its own guard:

$$\mathbf{S} \in Ev : S.\text{name} = \text{“}S\text{”}$$
$$\mathbf{R} \in Ev : R.\text{name} = \text{“}R\text{”} \vee (R.\text{name} \neq \text{“}R\text{”}).(R.\text{user} \neq S.\text{user})$$
$$\mathbf{P} \in Ev : (P.\text{name} = \text{“}P\text{”}).(P.\text{user} = S.\text{user}).(P.\text{t} < S.\text{t} + t_{out}),$$

where we consider as input an Ev(name, t, user) relation with fields for event name, event timestamp and associated user id, and we use . as a shorthand for the conjunctive operator. The pattern is described in terms of both selection predicates (for identifying the name of the event to be matched), as well as join predicates that make sure that the matched events are correlated by the id of the user that performed them and that the "Purchase" event occurs within a timeout t_{out} from the "Search" event.

The typical execution plan of this task on a map-reduce framework is to first group all events by user id, then sort them on time and finally run a pattern matching engine to detect the desired sequence of events. In this work we propose to greatly expand the array of possible execution plans by taking advantage of the fact that a large class of patterns can be equivalently expressed as SQL queries. By doing so then one can leverage decades of progress in query optimization to come up with more efficient query plans than the one outlined above. For instance, we can express our example pattern as a SQL query as follows:

```
SELECT S.*, P.*
FROM Ev S, Ev P
WHERE S.time < P.time
AND S.name == "S"
AND P.name == "P" AND P.time < S.time + t_out
AND P.user == S.user
AND NOT EXISTS (
    SELECT * FROM Ev NR
    WHERE NR.name != "R"
    AND S.time < NR.time AND NR.time < P.time
    AND NR.user == S.user );
```

The first part of the query enforces the fact that a successful match consists of an event S followed within a timeout t_{out} by an event P from the same user, while the second part captures the fact that only R events are allowed to take place in between these two events. The final result of the query is a set of tuples, one per successful match, consisting of the initial and final event in the match.

In the following we discuss our approach for performing this pre-processing step in a time and space efficient manner.

We start by defining for each event variable X of the pattern *symbolic sets* X^σ, which collect the values of X's fields that are joined by the query, based on the input events matching its selection predicates (for transition R, we get two such symbolic sets R^σ and $\overline{R}^\sigma$, as its guard has two disjoint selection predicates.).

$$S^\sigma := \{\langle S.\text{t}, S.\text{user}\rangle \mid S \in Ev : S.\text{name} = \text{“S”}\}$$
$$R^\sigma := \{\langle R.\text{t}\rangle \mid R \in Ev : R.\text{name} = \text{“R”}\}$$
$$\overline{R}^\sigma := \{\langle NR.\text{t}, NR.\text{user}\rangle \mid NR \in Ev : NR.\text{name} \neq \text{“R”}\}$$
$$P^\sigma := \{\langle P.\text{t}, P.\text{user}\rangle \mid P \in Ev : P.\text{name} = \text{“P”}\}$$

We introduce a slicing operator that simplifies our notation. Given a set S of tuples $\langle s_1, s_2, \ldots, s_n\rangle$ and unary predicates $\phi_1, \phi_2, \ldots, \phi_n$, we define the slicing operator:

$$\text{Slice}_{\phi_1,\phi_2,\ldots,\phi_n}(S) = \{\langle s_1, s_2, \ldots, s_n\rangle \mid \bigwedge_{i=1}^{n} \phi_i(s_i)\}$$

We will also represent $u_=(t)$ for the unary predicate that determines if t is equal to u, and $(l,u)_\epsilon(t)$ for the unary predicate that determines if t is in the (open) time interval (l,u).

Next, we re-write the query, first using comprehension syntax, and then using the slicing operator:

$$\begin{aligned} Q &:= \{\langle \mathtt{s},\mathtt{p}\rangle \mid \mathtt{s} \in S^\sigma, \mathtt{p} \in P^\sigma : \\ &\qquad \mathtt{p}.\text{t} \in (\mathtt{s}.\text{t} : \mathtt{s}.\text{t} + t_{out}) \ . \ (\mathtt{p}.\text{user} = \mathtt{s}.\text{user}) \ . \\ &\qquad \neg(\exists \mathtt{nr} \in \overline{R}^\sigma : \mathtt{nr}.\text{t} \in (\mathtt{s}.\text{t} : \mathtt{p}.\text{t}) \ . \ (\mathtt{nr}.\text{user} = \mathtt{s}.\text{user})) \ \} \\ &:= \{\langle \mathtt{s},\mathtt{p}\rangle \mid \mathtt{s} \in S^\sigma, \mathtt{p} \in \text{Slice}_{(\mathtt{s}.\text{t}:\mathtt{s}.\text{t}+t_{out})_\epsilon,\mathtt{s}.\text{user}_=}(P^\sigma) : \\ &\qquad \text{Slice}_{(\mathtt{s}.\text{t}:\mathtt{p}.\text{t})_\epsilon,\ \mathtt{s}.\text{user}_=}(\overline{R}^\sigma) = \varnothing \ \} \end{aligned}$$

Each symbolic set, in concert with the slice operator let us remove events from the input that are guaranteed not to participate in a successful match. For this example pattern we get the following filters:

$$\begin{aligned} \phi_S(\mathtt{s}) &\equiv \exists \mathtt{p} \in \text{Slice}_{(\mathtt{s}.\text{t}:\mathtt{s}.\text{t}+t_{out})_\epsilon,\mathtt{s}.\text{user}_=}(P^\sigma) : \\ &\qquad \text{Slice}_{(\mathtt{s}.\text{t}:\mathtt{p}.\text{t})_\epsilon,\ \mathtt{s}.\text{user}_=}(\overline{R}^\sigma) = \varnothing \\ \phi_R(\mathtt{r}) &\equiv \text{Slice}_{(-\infty:\mathtt{r}.\text{t})_\epsilon,\text{true},(\mathtt{r}.\text{t}:\infty)_\epsilon,\text{true}}(Q) \neq \varnothing \\ &\equiv \exists \mathtt{s} \in \text{Slice}_{(-\infty:\mathtt{r}.\text{t})_\epsilon,\text{true}}(S^\sigma), \\ &\qquad \exists \mathtt{p} \in \text{Slice}_{(\max(\mathtt{s}.\text{t},\mathtt{r}.\text{t}):\mathtt{s}.\text{t}+t_{out})_\epsilon,\mathtt{s}.\text{user}_=}(P^\sigma) : \\ &\qquad\qquad \text{Slice}_{(\mathtt{s}.\text{t}:\mathtt{p}.\text{t})_\epsilon,\ \mathtt{s}.\text{user}_=}(\overline{R}^\sigma) = \varnothing \\ \phi_{\overline{R}}(\mathtt{nr}) &\equiv \text{Slice}_{(-\infty:\mathtt{nr}.\text{t})_\epsilon,\mathtt{nr}.\text{user}_\neq,(\mathtt{nr}.\text{t}:\infty)_\epsilon,\text{true}}(Q) \neq \varnothing \\ &\equiv \exists \mathtt{s} \in \text{Slice}_{(-\infty:\mathtt{nr}.\text{t})_\epsilon,\mathtt{nr}.\text{user}_\neq}(S^\sigma), \\ &\qquad \exists \mathtt{p} \in \text{Slice}_{(\max(\mathtt{s}.\text{t},\mathtt{nr}.\text{t}):\mathtt{s}.\text{t}+t_{out})_\epsilon,\mathtt{s}.\text{user}_=}(P^\sigma) : \\ &\qquad\qquad \text{Slice}_{(\mathtt{s}.\text{t}:\mathtt{p}.\text{t})_\epsilon,\ \mathtt{s}.\text{user}_=}(\overline{R}^\sigma) = \varnothing \\ \phi_P(\mathtt{p}) &\equiv \exists \mathtt{s} \in \text{Slice}_{(\mathtt{p}.\text{t}-t_{out}:\mathtt{p}.\text{t})_\epsilon,\mathtt{p}.\text{user}_=}(S^\sigma) : \\ &\qquad\qquad \text{Slice}_{(\mathtt{s}.\text{t}:\mathtt{p}.\text{t})_\epsilon,\ \mathtt{p}.\text{user}_=}(\overline{R}^\sigma) = \varnothing, \end{aligned}$$

which when applied to their corresponding symbolic set will retain only those events that contribute to the output of Q. We detail the procedure for generating these precise filters from the relational query expressing the pattern in section 4.2.

Because evaluating these filters would in many cases be as expensive as computing the complete result Q, we use them only as a starting point for deriving a set of *abstract filters*, as a "relaxed" version of these *precise filters*, but that can be applied with low processing and communication costs. We do so by employing a series of *data and predicate abstractions* that generate conservative versions of the original filters.

We showcase our techniques on the filter corresponding to the S event variable, which highlights the fact that the Search events in the output are those that have only Read-review events by that same user between themselves and the next Purchase event occurring within a t_{out} window of time.

Data abstraction provides time and space efficient representations for symbolic sets S^σ and $\overline{R}^\sigma$. For example, one

could abstract over time and coarsen timestamps t into time intervals $\lfloor t \rceil$. Then, the time dimension of sets S^σ, $\overline{R}^\sigma$ could be efficiently encoded and queried as interval maps (i.e., bit vectors where each bit corresponds to a time interval and a set bit would denote the fact that an event has occurred within the corresponding interval). Using this abstraction the filter becomes:

$$\phi_S^{\lfloor t \rceil}(\mathtt{s}) \equiv \exists \mathtt{p} \in \text{Slice}_{\lfloor \mathtt{s.t}:\mathtt{s.t}+t_{out} \rceil, \mathtt{s.user}}(P^\sigma) : \\ \text{Slice}_{\lceil \mathtt{s.t}:\mathtt{p.t} \rceil,\ \mathtt{s.user}}(\overline{R}^\sigma) = \varnothing$$

where in order to maintain conservativeness (i.e., $\phi_S \rightarrow \phi_S^{\lfloor t \rceil}$) we must over-approximate the $\mathtt{s.t} : \mathtt{s.t}+t_{out}$ range but under-approximate the $\mathtt{s.t} : \mathtt{p.t}$ interval. Thus, in order to be sound (and, depending on the filter) data abstractions may provide both over- and under- approximations of the original sets.

There are many different ways to approach data abstraction. For example, hashing is one approach to abstract over sets. If we use a compact representation of sets S^σ, $\overline{R}^\sigma$ that stores time information only per user hash bucket as opposed to individual user ids, then the resulting abstract filter:

$$\phi_S^{\#user}(\mathtt{s}) \equiv \exists \mathtt{p} \in \text{Slice}_{(\mathtt{s.t}:\mathtt{s.t}+t_{out}), \#\mathtt{s.user}}(P^\sigma) : \\ \text{Slice}_{(\mathtt{s.t}:\mathtt{p.t}),\ \#\mathtt{s.user}}(\overline{R}^\sigma) = \varnothing$$

does not satisfy our safety requirement (i.e. $\phi_S \nrightarrow \phi_S^{\#user}$). This happens because hashing can only provide over approximations of sets while, in order to ensure conservativeness, the abstractions used for this filter need to provide both over and under approximations. While there are several ways to address this issue (for an alternative solution see section 4.3), in the following we show how *predicate abstraction* can alleviate the problem.

Predicate abstraction encompasses a set of re-writings that relaxes the filter by discarding those constraints that cannot be safely or efficiently abstracted over. For instance, our filter could be weakened into:

$$\psi_S(\mathtt{s}) \equiv \text{Slice}_{(\mathtt{s.t}:\mathtt{s.t}+t_{out}),\ \mathtt{s.user}}(P^\sigma) \neq \varnothing,$$

which eliminates only those search events that are not followed by a purchase event within t_{out}, and was obtained from the base case of the existential quantifier in ϕ_S. Since this version only requires over-approximation, one can safely use hashing to abstract over the user id dimension of P^σ, i.e. $\psi_S \rightarrow \psi_S^{\#user}$, where

$$\psi_S^{\#user}(\mathtt{s}) \equiv \text{Slice}_{(\mathtt{s.t}:\mathtt{s.t}+t_{out}),\ \#\mathtt{s.user}}(P^\sigma) \neq \varnothing.$$

Moreover, predicate abstraction also reveals the well known Bloom join algorithm as an instance of our approach, wrt. the join predicate $P.\text{user} = S.\text{user}$ from the original query. This becomes apparent if we ignore time in the filter above:

$$\psi_S^{\#user}(\mathtt{s}) \equiv \text{Slice}_{*,\#\mathtt{s.user}}(P^\sigma) \neq \varnothing,$$

and we use a Bloom filter to implement $\text{Slice}_{*,\#\mathtt{s.user}}(P^\sigma)$. More importantly, it highlights the fact that one can use data and predicate abstraction to explore the entire spectrum of abstract filters, and make the choice between precision vs overheads on a case by case basis. We discuss additional scenarios where predicate abstraction proves beneficial in section 4.4.

3. RELATED WORK

Complex event processing has received extensive interest in the literature [5, 9–11, 17, 26] and has enjoyed similarly wide adoption in industry [1–4, 13], with most of the work focused on online (near-real time) systems designed to deliver extremely low response times for mission-critical tasks (for eg., blocking a credit card upon detecting fraudulent activity). However, these systems experience scalability issues as the volume of data that needs to be processed continues to grow and considering that the pattern matching logic is typically implemented in a sequential manner. Moreover, the optimizations that they deploy focus primarily on computation reuse and sharing [25], and thus are not appropriate for addressing the demand for more parallelism.

In dealing with the increase in load, most systems dismiss, as early as possible, of all the input events that fail to satisfy the selection predicates of at least one of the variables in a given pattern. A more aggressive **preprocessing** step has been proposed by Cadonna et. al. [12], which leverages both the structure of the pattern and the guards of its variables to formulate a set of necessary conditions for a window of events to contain a complete match. If a particular window fails to meet those conditions its can be ignored, thus completely sidestepping the pattern matcher. While their proposal targets only patterns specified as *sequenced event sets* [11], we support arbitrary automata, and for a large class we derive both necessary and sufficient (i.e. precise) conditions for identifying complete matches. More importantly, we discuss how the precision of these filters can be traded in favor of low overheads by proposing *data and predicate* abstractions, and we showcase their ability to minimize data shuffling when performing pattern matching over a map-reduce framework.

Solutions for **distributed/parallel pattern matching** have focused mainly on three directions: a) partitioning the input, either by time windows [22] or key attributes [19], and then processing each partition sequentially, b) partitioning the pattern via query plans [6] that produce the output from progressively larger sub-patterns and where the intermediary results are mined on distinct processing nodes, and c) partitioning the set of partial matches/runs that need to be explored at any given time [7]. The first approach is vulnerable to data skew, in case of large time windows or attribute keys with few distinct values, while the second one does not support the Kleene star. More importantly, all of them target only the pattern matching process itself and disregard the communication costs incurred when the analysis is performed on a map-reduce platform. By contrast, our technique of *abstract pattern matching* limits the number of events that need to be considered by the pattern matching operator in the first place, and as such can even be used in conjunction with the proposals described above. Moreover, in our work we expose the inherent parallelism of patterns by expressing them in terms of embarrassingly parallel relational operators.

Symbolic execution [21] has been proposed as a way to speedup user-defined aggregates [24], and in particular pattern matching routines executing within the reduce stages of map-reduce workloads. The user defined aggregate is symbolically evaluated on each partition of the input and only the symbolic summary of the execution is submitted to the reducer. The reducer can then determine the final result based on the collected symbolic summaries. This approach ends up shuffling in many cases significantly less data between mappers and reducers as the symbolic summaries are

typically much smaller than the original input. However, as it requires that the data be already ordered by time, it is not applicable to the common scenario where workload wide considerations impose a different physical layout for the input data. Moreover, even if the ordering constraint on the input holds, this approach can still leverage *abstract pattern matching* to address the pathological scenarios when the symbolic summaries are just as large as the original input, as it would allow it to more accurately prune unfeasible paths based on a global view of the domain of join attributes.

Proposals for coping with **out-of-order event series** have primarily focused on buffering mechanisms and efficiently updating the internal structures of the pattern matcher in response to events arriving late [15, 20, 23]. Since they operate under the assumption that misplaced events are relatively rare (i.e. the input stream is mostly ordered), use time thresholds beyond which delayed events are simply ignored, and rely on a complete view of the input, they cannot mitigate the expensive data shuffling that precedes pattern matching on map-reduce platforms.

4. DESIGN

Evaluating pattern matching queries in a map-reduce framework usually adds a reduction step in order to sort the input, which can become the main bottleneck of the workload, both at the network level (large amounts of shuffled data) and at the processing level. The standard approach to minimize the cost of sorting/data shuffling has been to introduce a *preprocessing* phase which first filters the input based on the *selection* predicates, i.e. removes all events that do not satisfy the guard of at least one event variable while ignoring its *join* predicates. This *preprocessing* phase can significantly reduce both processing costs and latency since it takes linear time in the size of the input (as opposed to $O(nlogn)$ for sorting) and is embarrassingly parallel, i.e. scales out with the number of computing resources available. Moreover, it can be merged with the previous operator in the data processing pipeline thus incurring no extra costs for materialization or data transfer.

In our work we extend the opportunities for query plan optimizations across all the stages of the workload, beyond just pipelining the preprocessing phase of the pattern matcher, by leveraging the fact that a large class of patterns can be equivalently expressed as relational queries. Whenever that is not the case we can soundly narrow the scope (i.e., through conservative predicate abstraction) of our optimizations to the sub-patterns that do. The resulting relational expressions can then be optimized within the scope of the entire (predominantly relational) workload based on decades of progress in relational optimizations.

Even in the scenarios where the relational optimizer decides that using the pattern matcher leads to the most efficient query plan, we can leverage the relational expressions to generate a *precise filter* which retains only those input events that appear within a complete match. It achieves that by fully exploiting the pattern's structure along with its *join* predicates (i.e., user ids in our motivating example), as opposed to just the *selection* predicates (i.e., the unary transition predicates). Applying the precise filter as part of the preprocessing step leads to a dramatic improvement in its reduction ratio. This is unsurprising considering that the number of events forming complete matches is usually tiny compared to the input stream's size.

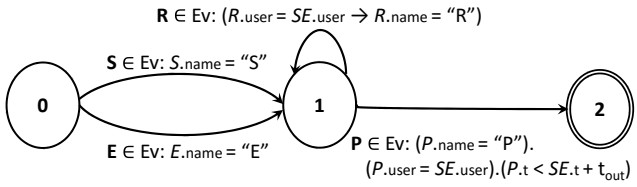


Figure 2: Automata for the (S|E)R*P pattern.

However in many cases the precise filter would be too expensive to build and evaluate as such. Therefore, we coarsen it to obtain an *abstract* filter which can be constructed and queried in a time and space efficient manner. In particular, we make use of both *data* and *predicate* abstraction in order to generate a filter that, while conservative, closely matches the *precise* filter. Thus, in many cases we manage to discard most of the events that are guaranteed not to take part in a successful match and significantly reduce the amount of data fed into the pattern matcher. We explore the trade-offs between the overheads incurred in building/querying the filter and its accuracy.

The derivation of abstract filters is not strictly tied to the ability to generate a semantically equivalent relational expression for a pattern. By adding a fixpoint construct, we demonstrate analogous techniques for generating both the *precise* and *abstract* filters.

4.1 From patterns to relational queries

To simplify the presentation we detail our approach on patterns specified as a finite automata where each transition is annotated by an event variable and a guard. Turning regular expressions-like patterns to automata is straightforward, and in many cases there is a direct mapping between the pattern's event variables and the automaton's transitions. Figure 2 shows the automaton corresponding to the (S|E)R*P pattern, where S,R, and P, correspond to the same "Search", "Read review" and "Purchase" events described in section 2, and E denotes the event of responding to a promotional email. In the guards of the pattern, the SE variable references either the S or E variable depending on the event that initiated the current match.

We formally define a finite automaton as $\mathcal{A} = (S, T, s_{start}, C)$, where S is the set of states, T is the set of transitions, s_{start} is the initial state and C is the set of completion (accepting) states. Each transition is defined in terms of the tuple (X, p_X, src, dst) where X is the variable binding the event currently considered by that transition, p_X is the guard (a propositional formula) deciding whether the transition can be triggered or not, and src and dst are the transition's source and destination states. The atomic formulas of the guard are either *selection* predicates, which only reference the variable associated with the current transition, or *join* predicates, which may also reference the variables of preceding transitions. In particular, the guards of start transitions can only use selection predicates.

The translation to relational expressions are not limited to acyclic automata, but also to automata with cycles of fixed length, i.e. each iteration of the cycle has the same number of transitions. This class of automata is of particular importance as it covers the vast majority of patterns found in benchmarks and in industrial workloads.

Notation. We usually denote states by indices i, j, k, and

we abuse notation to refer to transitions using the variable name they introduce (eg. X, Y, Z). In addition we refer to states and transitions also as nodes and edges, respectively, in the corresponding graph of an automaton.

To streamline the presentation we begin by considering automata with only cycles of length 1 and we distinguish between cycle transitions and non-cycle transitions.

The translation process produces one relational query Q_i per state i, and the final relational expression of the automata is obtained by unioning all the queries generated for the automata's accepting nodes. Evaluating query Q_i over a set of events computes partial matches, i.e. sequences of events, corresponding to all the possible paths between the starting node and node i. Therefore, if i is the starting node then its query returns an empty sequence while if i is an accepting node then it returns complete matches found in the input stream of events. Moreover, the partial (complete) matches computed by Q_i do not include the events matched against cycle transitions therefore have a bounded length.

The schema of the queries we generate consist of a sequence of event variables, one for each non-cycle transition that may occur along its associated set of paths. If a transition is triggered within a partial match, then its corresponding variable is initialized by the event that triggered it, otherwise that variable is assigned null. For our example, the schema of query Q_2 consists of S, E and P, and for each of its output tuples either S or E is set to null.

State queries Q_i are defined as the union of transition queries Q_X over all the incoming non-cycle transitions into state i, where each transition query Q_X computes partial matches corresponding to the paths ending with transition X. In turn, the Q_X query corresponding to a non-starting, non-cycle transition is defined as the join between node query Q_k, where k is the source of the transition, and the input relation of events, where each event considered is bound by variable X. The condition enforced by Q_X consists of the guard p_X along with the constraint that the timestamp of X succeeds the last event in the partial match produced by Q_k. Additionally, a nested query ensures that no other events exist between the last event matched by Q_k and the event bound by X, except for events matching cycle transitions starting and ending in k. By contrast, for starting transitions we only need to apply the transition's guard p_X over the input relation.

The translation process iterates in topological order over the nodes of the DAG obtained by ignoring the cycle transitions of the automaton. At each node i, it first generates the queries for all its incoming non-cycle transitions Q_X and then Q_i is defined as their union. The schema of Q_i is established as the union of the schemas of the incoming transitions Q_X.

Applying the procedure outlined above to our example produces the following queries:

$$Q_S = \{S \mid S \in Ev : p_S\} \quad Q_E = \{E \mid E \in Ev : p_E\} \quad Q_1 = Q_S \cup Q_E$$

$$Q_P = \{\langle SE, P\rangle \mid SE \in Q_1, P \in Ev : p_P \;.\; (SE.\mathrm{t} < P.\mathrm{t}).\; \{R \mid R \in Ev : R.\mathrm{t} \in (SE.\mathrm{t}, P.\mathrm{t}) \;.\; !p_R\} = \varnothing\}$$

$$Q_2 = Q_P$$

Translating multi-transition cycles of fixed length (≥ 1) to relational queries requires that we first normalize them such that each cycle has a single starting node and a single ending node, and that the two coincide. A starting node for a cycle is defined as the destination of one of its incoming edges, while an ending node is the source of one of its outgoing edges. Cycles of fixed length cannot have transversal edges (paths), i.e. edges (paths) that connect non-adjacent nodes in the cycle, as this would violate the restriction that each of the cycle's iterations has the same length.

Given an automaton with a cycle that has multiple starting and ending nodes we first duplicate the cycle for each additional starting node. Then, for the resulting cycles we duplicate the path between their starting node and their last ending node (i.e. the furthest from the starting node). Finally, we change the source of each outgoing edge to the corresponding node in the newly created path, resulting in a cycle whose incoming and outgoing edges have the same node as destination and source, respectively. By applying this procedure to every cycle with multiple starting and ending nodes we obtain a normalized automaton. While the resulting automaton may have multiple cycles starting and ending with the same node, all of those must also have the same length.

The only part of the translation process that changes when generalizing from automata with single edge cycles to normalized automata is the specification of non-cycle transition queries Q_X, and in particular, the specification of its nested query should the source state k of X be the starting/ending point of a cycle. We recall that in the case of cycles with a single transition Y the nested query enforces that all events occurring in the interval between the last event in the partial match computed by Q_k and the timestamp of event variable X satisfy guard p_Y. By contrast, in the case of multi-transition fixed length cycles, for each event in the same interval we establish its position (based on the count of events with smaller timestamps) and we ask that it satisfies the guard of the transition corresponding to that position in the cycle modulo the length of the cycle. If multiple cycles initiate and conclude at the same node, we alternatively have to enforce that all events in an iteration satisfy the corresponding transition guards of a particular cycle.

4.2 Precise filter generation

After translating patterns into relational queries a host of relational optimizations become applicable, from column pruning and partial aggregation to the selection of specific join algorithms. In the following we detail our proposal for speeding up pattern matching in a distributed environment based on its representation in the relational world as a series of unions and joins. The first step in this process is to derive a *precise* filter which retains from the input relation only those events guaranteed to appear in a successful match. While it is understood that constructing and applying such filters may prove too expensive to evaluate directly, we discuss them nonetheless as they are essential in guiding the design of the *abstract* filter, its time and space efficient variant.

The precise filter of an automaton $\mathcal{A}$ consists of multiple components, one for each of its transitions, and an input event is rejected if it does not satisfy any of these components. In current work we derive precise filters only for non-cycle transitions and single-edge cycles, since the filters for transitions in multi-edge cycles are impractical to build/apply and abstract over, as they require the position of the considered event within a particular time interval.

The precise filter ϕ_X corresponding to a non-cycle tran-

sition X of automaton $\mathcal{A}$ is extensionally defined in terms of the events from the input that bind the event variable X in the output of its semantically equivalent relational query $Q_{\mathcal{A}}$. Therefore ϕ_X can be obtained from the definition of $Q_{\mathcal{A}}$ by projecting away (i.e. existentially quantifying) all the other event variables in its output besides X. In our running example the precise filter derived for transition P is:

$$\phi_P(P) \equiv \exists SE \in Q_1 : p_P \ . \ (SE.\mathrm{t} < P.\mathrm{t}) \ . \\ \{R \mid R \in Ev : R.\mathrm{t} \in (SE.\mathrm{t}, P.\mathrm{t}) \ . \ !p_R\} = \varnothing$$

The precise filter ϕ_Y of a cycle transition Y, with node k as source and destination, selects from the input those events that occur within interval (t_Z, t_W), where t_Z, t_W, are the timestamps of a pair of event variables Z, W, from the output of $Q_{\mathcal{A}}$ such that Z, W are associated to non-cycle transitions entering and respectively exiting k. ϕ_Y does not need to enforce the guard p_Y as it is guaranteed that all events between t_Z and t_W satisfy p_Y based on the nested query generated as part of the definition of Q_W (and which was found to hold during the evaluation of $Q_{\mathcal{A}}$). For the cycle transition R in our example we generate the following filter:

$$\phi_R(R) \equiv \exists \langle SE, P\rangle \in Q_{\mathcal{A}} : R.\mathrm{t} \in (SE.\mathrm{t}, P.\mathrm{t})$$

We take a bottom-up approach to building the filters as it allows us to outline an evaluation strategy that operates over sets and which uses set operations like union, intersection, set membership or emptiness testing. Adopting such a set-centric evaluation strategy is advantageous in a distributed environment due to the embarrassingly parallel nature of many set operators, but more importantly it gives us a powerful knob in terms of the set representations that we use, making it possible to trade off precision in favor of performance. This strategy is what ultimately guides the design of *abstract* filters (discussed in sections 4.3 and 4.4), which make our solution practical.

As a first step we build a *symbolic set* X^σ for every transition variable X of an automaton $\mathcal{A}$, which collects the values of X's fields joined throughout all the transition guards of $\mathcal{A}$, as X is bound to the input events that satisfy p_X's selection predicates. We then re-write the precise filters by replacing each event variable and selection predicate associated to a transition with its corresponding symbolic set. Finally, the join predicates get re-written in terms of slicing, intersection and emptiness testing over these sets.

The process is showcased in section 2 where we derive symbolic sets S^σ, R^σ and P^σ, which we then use in the definition of precise filters ϕ_S, ϕ_R and ϕ_P along with slicing operators that express the join predicates of the pattern. We omit the minute technical details of turning transition guards into expressions over symbolic sets as they are not particularly challenging. It involves turning the guards into disjunctive normal form, and replacing in each conjunct the transition variables and their selection predicates with the corresponding symbolic set, and finally using slicing to encode their join predicates.

4.3 Data abstraction

Precise filters provide a template for designing sound and conservative set abstractions. Such abstractions should never eliminate events that *could* contribute to a successful match.

The sets we abstract over contain tuples as opposed to single values, where each tuple field holds the values relevant to a particular join predicate. Similarly, the abstractions we chose need to be multidimensional in the sense that they must allow the testing of the domain of values corresponding to a specific join predicate, independent of the others. Second, we note that besides supporting set intersection and set union, the choice for a particular set abstraction is deeply influenced by the particular kind of predicates said abstractions needs to support. For example, in the case of equality joins an appropriate data abstraction would be to use Bloom filters as they provide a low cost solution for testing whether o value belongs to a set, with the guarantee of no false negatives. For enforcing inequality joins on the other hand, an interval map, i.e. a bit vector where each bit stands for a particular interval in the domain, provides a similarly low cost abstraction.

Finally, depending on whether the symbolic sets are tested for non-emptiness or emptiness, their abstraction has to provide either an over- or an under-approximation of the original contents. For instance, when abstracting over the precise filter ϕ_S from the example in section 2, we have to use over-approximating set abstractions for S^σ, while the opposite is true for $\overline{R}^\sigma$. While this is of no concern for set abstractions, like interval maps, which support both, it does pose a challenge to abstractions based on hashing, like Bloom filters, which can only provide over-approximations.

One way to address the issue raised by hash-based data abstraction wrt. sets that require under-approximation is to replace them with the coarsest under-approximation possible, i.e. the empty set $\varnothing$ (as showcased in section 2). Alternatively, one can re-write the precise filter using min aggregates:

$$\omega_S(\mathtt{s}) \equiv \min\{\mathtt{p}.\mathrm{t} \mid \mathtt{p} \in \mathrm{Slice}_{(\mathtt{s}.\mathrm{t}:\mathtt{s}.\mathrm{t}+t_{out}),\ \mathtt{s}.\mathrm{user}}(P^\sigma)\} \\ < \min\{\mathtt{r}.\mathrm{t} \mid \mathtt{r} \in \mathrm{Slice}_{(\mathtt{s}.\mathrm{t}:\mathtt{s}.\mathrm{t}+t_{out}),\ \mathtt{s}.\mathrm{user}}(\overline{R}^\sigma)\},$$

which captures the fact that a Search event is part of a complete match if the next Purchase event precedes the next event different from Read-review. Now, we can safely use hashing to abstract over user ids as follows:

$$\omega_S^{\#user}(\mathtt{s}) \equiv \\ \min_{u \in \#s.\mathrm{user}} \min\{\mathtt{p}.\mathrm{t} \mid \mathtt{p} \in \mathrm{Slice}_{(\mathtt{s}.\mathrm{t}:\mathtt{s}.\mathrm{t}+t_{out}),\ u}(P^\sigma)\} \\ < \max_{u \in \#s.\mathrm{user}} \min\{\mathtt{r}.\mathrm{t} \mid \mathtt{r} \in \mathrm{Slice}_{(\mathtt{s}.\mathrm{t}:\mathtt{s}.\mathrm{t}+t_{out}),\ u}(\overline{R}^\sigma)\}.$$

While for this example it was possible to come up with an alternative formulation of the precise filter, this may not always be the case. And even if it is possible, the alternatives may be too expensive to materialize and query (for eg. evaluating $\omega_S^{\#user}$ can easily be more costly than $\phi_S^{\lfloor t \rfloor}$). Therefore, in the next section we discuss how *predicate abstraction* can mitigate these kinds of issues.

4.4 Predicate abstraction

We propose *predicate abstraction*, ie. the technique of weakening the precise filters by discarding some of their predicates, as a way of overcoming the challenges that can arise when turning them into abstract filters. For example, it may happen that for some predicate types (for eg. $x.Contains(y)$, where x, y are strings) we simply cannot provide any data abstraction, and even for those that we can, materializing and querying those data abstractions might prove too expensive.

Predicate abstraction is an essential component of our approach allowing us to strike the right balance between the data reduction that the abstract filters provide on one hand,

and the overheads introduced by their data abstractions on the other. For instance, we may choose to discard predicates that have very low selectivity, i.e. the reduction in input data that they provide does not justify the cost of enforcing them. Similarly, one may turn to predicate abstraction when dealing with patterns with a large number of join predicates or transitions, in order to mitigate the increased overheads incurred by their data abstractions.

Given a join predicate $\theta(X.f, Y.f)$, the definition of precise filter $\phi_X(\mathtt{x})$ is bound to contain a corresponding slicing of Y's symbolic set as $\text{Slice}_{\ldots,f}(Y^\sigma)$. Just like in the case of data abstraction, depending on whether predicate θ appears in a negated sub-clause or not, its abstraction needs to be under or over approximating, in order to remain sound. More precisely, we enforce the over-approximation of θ by assuming that the f dimension of Y^σ is invariably the entire domain of f. Analogously, the under-approximation of θ leads to the assumption that the f dimension of Y^σ is invariably void, which by consequence reduces the entire Y^σ to the empty set. In section 2, it was the under-approximating abstraction of predicate $S.\text{user} = R.\text{user}$ that produced the relaxed filter ψ_S, by turning $\overline{R}^\sigma$ in ϕ_S to the empty set.

Finally, predicate abstraction need not be applied symmetrically, i.e. given join predicate $\theta(X.f, Y.f)$ one could choose to abstract over X^σ's f dimension but not over the f dimension of Y^σ, or vice-versa. This can prove useful if one of the symbolic sets is known to be a subset of the other, thus one would need to build an abstract filter only for the smaller one. Moreover, considering that initial or final transitions typically have the fewest number of matching events, one might choose to only collect and abstract over the symbolic sets of those transitions. Due to their low cardinality these sets are likely to have very high filtering power and the data abstractions used to enforce them can achieve higher precision for the same operating costs (considering the small number of values to store and query). Applying this heuristic to our example from section 2 wrt. to the final transition P (as we expect to have relatively few purchasing events) produces the following relaxed filters:

$$\gamma_S(\mathtt{s}) \equiv \text{Slice}_{(\mathtt{s.t}:\mathtt{s.t}+t_{out}),\mathtt{s.user}}(P^\sigma) \neq \varnothing$$
$$\gamma_{\overline{R}}(\mathtt{r}) \equiv \text{Slice}_{(\mathtt{r.t}:\infty),\mathtt{r.user}}(P^\sigma) \neq \varnothing$$
$$\gamma_P(\mathtt{p}) \equiv \mathtt{true},$$

as obtained from the definitions of $\phi_S, \phi_{\overline{R}}$ and ϕ_P by replacing S^σ with the full domain of time/user ids and $\overline{R}^\sigma$ with the empty set. By further applying data abstraction to these filters, we end up with a relatively cheap way to dispose of a large number of the input events, irrelevant to our pattern.

4.5 Building filters through fixpoint

Abstract filters are not strictly tied to the expressibility of automata as relational queries, but that they can also be computed for an arbitrary automaton by using a fixpoint operator that keeps track of provenance information. The fixpoint operator we consider assigns to the starting node the empty sequence and then iteratively builds for every node of the automaton its corresponding set of (partial) matches, until no more new matches are found. In every iteration, for each node i and each of its outgoing transitions X, the partial matches added by the previous round to i get extended if matching events are found within the symbolic set of X. The newly found matches then get added to the collection of matches corresponding to X's destination, and the process starts over. Since we never remove partial matches we are guaranteed to reach a fixpoint. Based on the complete matches collected by the accepting nodes, we also update the abstract filter's components corresponding to the transitions along their path.

In designing the fixpoint operator we similarly make extensive use of sets and set operators. In particular, the (partial) matches we compute for each node i of the automaton are represented as a sequence of sets, one for each transition that may occur on a path from the start node to i. Then the abstract filter for a specific transition can be obtained by unioning its corresponding set across all the accepting nodes. When deciding whether a partial match can be extended by triggering transition X, we simply intersect the symbolic set X^σ with the projection from the partial match of the corresponding join fields, i.e. those fields of previously occurring transitions that are joined against X. If the result is empty then the extension is not possible, otherwise a new partial match gets created whose corresponding set for transition X is the result of the intersection.

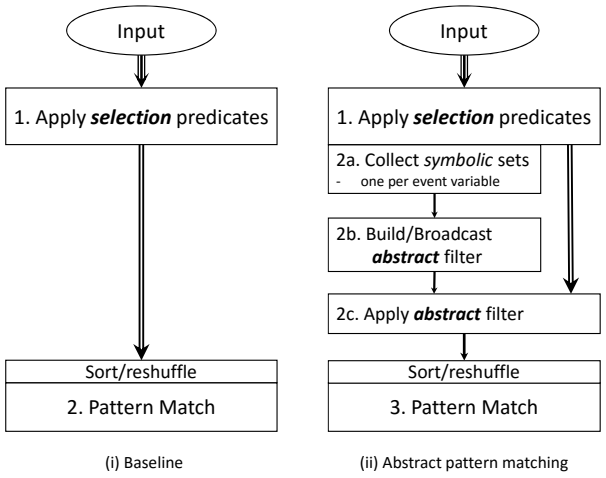


Figure 3: Map-Reduce execution plans for pattern mining using: (i) the Baseline approach or (ii) Abstract Pattern Matching.

5. IMPLEMENTATION

In the following we detail how we implemented our solution for speeding up large scale event series pattern matching on top of the Cosmos/Scope [14] map-reduce framework. We recall that the standard way of performing event-series pattern matching in such frameworks is to first remove all the events that do not match any of the selection predicates specified by the pattern, and then to sort on time/reshuffle the remaining events and finally process them using a pattern matching engine (see figure 3(i)).

Our solution significantly extends the amount of data that gets filtered out during the preprocessing stage by constructing an *abstract* filter which also enforces the join predicates occurring in a pattern (as opposed to just the selection predicates). The execution plan that we propose introduces 3 intermediary steps as is outlined in figure 3(ii). The first one collects in a parallel fashion a symbolic set for each event variable of the pattern (step 2a), followed by the union of these sets across all partitions of the input, and the re-

sults are then used to build the abstract filter by enforcing the join predicates between event variables (step 2b). We then broadcast the abstract filter back to the preprocessing nodes and have them apply it over the output of the initial selection-predicate based filter (step 2c). Finally, the remaining events are sorted/reshuffled, just like in the standard approach and processed by the pattern matching engine.

In implementing the execution plan in figure 3(ii), we make use as much as possible of the relational constructs and annotations provided by the Scope query language in order to maximize the potential for optimizations at the level of the entire data processing pipeline (for eg., we use native Scope to extract the event fields used in constructing the symbolic sets as well as to apply the filters that we build). For operating with the set abstractions themselves we use the rich extensibility features of Scope, all the while providing hints to its query optimizer.

Even though the *abstract* filters have the potential to dramatically reduce the amount of data that gets sorted/reshuffled /pattern matched, constructing them introduces overheads both in terms of processing costs and latency. In terms of processing costs we note that, while the construction of the symbolic sets can be done in the same pass as the selection-predicate based filtering, for applying the abstract filters we need a second pass over the data. Nonetheless, since applying a filter is linear in the size of the input, this extra pass ends up being much cheaper than sorting on time the same input, and if the reduction ratio of the filter is considerable then applying it results in a net win.

In terms of latency, our approach also introduces an extra reduction phase as required for aggregating the symbolic sets computed on each partition to a particular node where we can propagate the constraints specified by join predicates in order to obtain the abstract filters. This is also followed by a broadcast phase that delivers the abstract filters back to the nodes processing the input stream. In order to minimize the penalty in latency incurred by these two steps we limit the size of the abstract filters to the order of megabytes and we exploit the algebraic properties of the operators of the set abstractions that we use in order to optimize the aggregation (i.e. we make use of a *recursive* user defined aggregates).

6. EVALUATION

We tested our approach on 2 workloads: i) an internal Microsoft production system, consisting of 15 patterns (A1-15) matched over telemetry events collected within a 2 hour interval, and ii) GitHub, performing repository analytics (patterns G1-3) over the GitHub dataset consisting of events collected over a 5 years period. All transitions in a pattern have a main key constraint, i.e. all events in a match should belong to the same join condition for the Microsoft workload, or the same repository for the GitHub patterns, and a time constraint, i.e. all events considered should occur within a timeout from the initial event in the match. In addition, 9 patterns also have a secondary key constraint between some of their transitions. The characteristics of our workload are summarized in figure 4. The queries that we experiment with have up to 25 transitions and make use of both *union* and *Kleene star*. In particular, queries A10 and A15 apply the Kleene star over a sub-pattern as opposed to a single query variable.

We experimented on a virtual cluster consisting of 85 nodes as part of the Cosmos [14] infrastructure. We assess the benefits provided by our approach in terms of the ratio of input events vs selected events, the total execution time across the processing nodes of the cluster, and the time it takes to complete the query (latency). In measuring the data reduction provided by our abstract filters for a specific query we use as baseline only the events from the input stream that satisfy the selection predicates of some transition in the query. This way we can evaluate our approach in isolation from the standard technique of pushing selection predicates into the scan operator of a query. The resulting amount of data to be processed by each query is detailed in figure 4. Finally, we use query A5 to highlight the individual filtering potential of different join predicates, as well as their sensitivity wrt. the amount of state dedicated to their abstract representation.

Note that we also do not measure the cost to run the pattern matching engine and so our overall system processing time and latency numbers are conservative estimates of a production installation of our system.

6.1 Processed data reduction

In order to establish the raw potential of our approach we look at the amount of data that is discarded by the abstract filters that we build. The baseline consists only of events whose type matches the event type of a transition in the query.

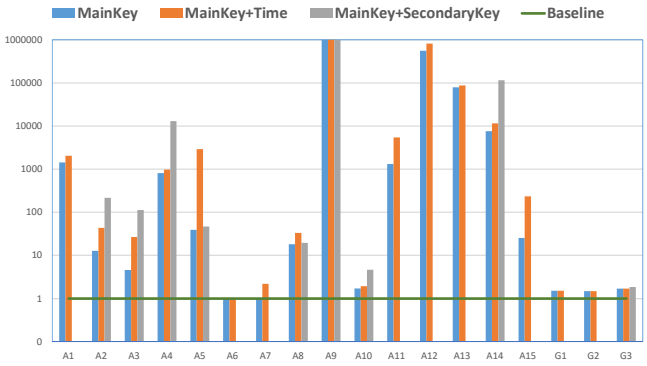
Figure 5a shows on a logarithmic scale the ratio between the data processed by the baseline approach and the data processed by our solution when considering different join predicates for building the abstract filters. In particular, we look at three types of join predicates: those referencing the main join key of the query (MainKey), those imposing constraints on the timestamp of matching events (Time), and those referencing a secondary join key (SecondaryKey). We report the results provided by the MainKey filter as well as its combination with the Time and SecondaryKey filter (for queries with joins on a secondary key).

For 12 out of the 18 queries in our workload we obtain at least a 10x reduction in the amount of data that has to be considered by the pattern matcher, with 3 of them ending up processing 5 orders of magnitude less data. The other 6 queries exhibit modest or no data reduction. This is mainly due to the precision lost by our abstract filters, as well as the fact that not all join predicates from a query can be efficiently abstracted over.

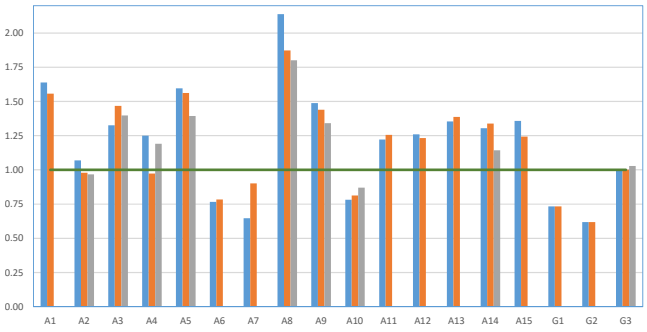
While incorporating extra join predicates in the abstract filters leads to further savings, it is dependent on the query and the stream of events which additional join predicate would provide the most benefits. In our workload we observe that adding the Time filter for queries A5 and A8 provides the biggest improvement while for queries A2-4, A10, A14 and G1 the SecondaryKey filter is more advantageous. Due to this variability one has to decide on a query by query basis which join predicates to use and how much state to allocate for them within the abstract filter. For our workload we assign to the MainKey, Time and SecondaryKey filters between 16KB and 4MB, 8B and 180B, and 2B and 8B, respectively. This allocation has been chosen under the constraint of a total size for the abstract filter in the order of megabytes and while considering the particularities of queries, for example, we take into account the fact that a query with a large timeout window would not benefit from a finegrained Time filter.

Query	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	G1	G2	G3
No. of Transitions	2	3	3	2	3	2	3	5	6	25	4	5	3	13	18	3	4	2
Has Union									✓	✓		✓		✓	✓	✓		
Has Kleene Star									✓	✓ (group)	✓	✓	✓	✓	✓ (group)		✓	
Has SecondaryKey		✓	✓	✓	✓			✓	✓	✓				✓				✓
Input Size(GB)	216	321	477	264	1361	180	148	158	1167	587	234	1167	286	366	141	593	593	85

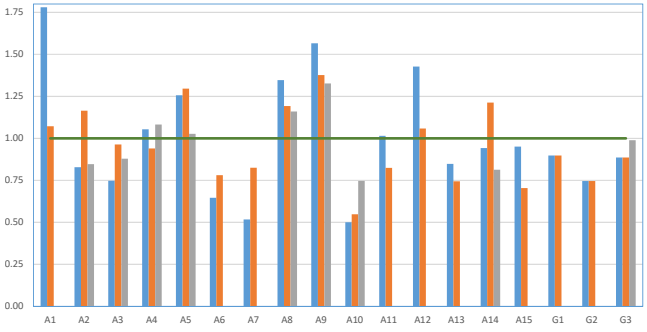
Figure 4: Workload characteristics



(a) Processed data reduction



(b) Processing time speedup



(c) Latency speedup

Figure 5: Reduction ratios of processed data and processing speedups when using abstract filters wrt. the corresponding baseline measures.

6.2 Abstract filter size vs reduction ratio

We vary the size of the abstract filter that we build as well as its configuration in order to asses the impact on the reduction ratio it provides. In particular, we experimented for query A5 with filters of sizes between 1024 to 8192KB and with a granularity for the MainKey filter between 65K to 2M distinct values. For each abstract filter size and MainKey granularity we devote the rest of available space to either the Time or the SecondaryKey filter. Since the filters we build are multiplicative in their composition, the granularity afforded to the Time and SecondaryKey filters varies between 4 and 1024 distinct values depending on the granularity of the MainKey filter as well as the total size of the abstract filter. In addition we record for reference the reduction in data achieved just by the MainKey filter for its different configurations.

From the results presented in figure 6 we conclude that for query A5, MainKey+Time is the most effective configuration as it provides the most reduction in the number of input events irrespective of the other parameters of the abstract filter. As expected, the bigger the size of the abstract filter the larger is the reduction rate achieved. We also note that the distribution of state amongst the components of the abstract filter is also important, as refining the granularity of only one of its components while keeping the total size of the filter fixed can experience a sweetspot beyond which the reduction rate is negatively impacted. This is reflected in our results for the MainKey+Time configuration where the 512K granularity setting for the MainKey component outperforms the 2M setting across all filter sizes. The drop in the reduction rate is a consequence of the fact that the MainKey filter reaches a plateau, beyond which increasing its granularity no longer increases the reduction rate, while decreasing the granularity of the Time filter is bound to decrease its filtering power.

There is also a knock-on effect between the different components of the abstract filter, as a poor choice for the granularity of one component can inhibit the filtering potential of the other components as well. For example, in query A5 a too small granularity for the MainKey filter (65K setting in figure 6) results in poor performance for the SecondaryKey component as well, as it cannot cope with the large number of events and their associated secondary keys it needs to discriminate between. Notably, the Time component is less susceptible to this issue.

6.3 Total processing time speedup

In the following we look at the total processing cost across all nodes in the cluster. This is a particularly relevant metric for cluster setups that allow workload consolidation or data centers that charge users based on total number of "processing hours" consumed. Figure 5b shows speedups of 1.25x to 2.14x for 11 out of the 18 queries, while 5 patterns experience slowdowns of at most 0.62x. The slowdowns are a consequence of performing an additional pass over the input in order to build the abstract filters, with little or no benefit in terms of discarded events.

In order to highlight the tradeoffs of our approach we break down the *baseline* execution of a query into the time it takes to i) read and sort the data, and ii) perform the re-

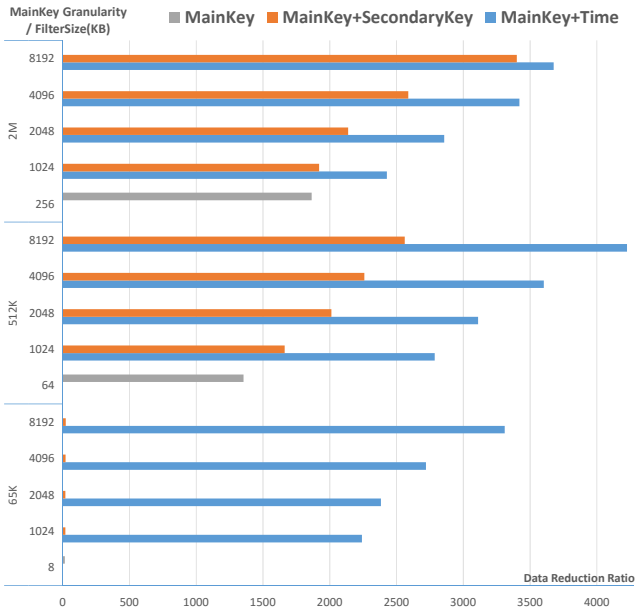


Figure 6: Tradeoff for query A5 between filter size (1024KB-8192KB) and data reduction for different filter configurations and with different granularities for the MainKey filter (65K,512K,2M).

duction step. On the other hand for our technique we look at the time it takes to i) read the data, ii) build the symbolic state associated with each transition, iii) reduce the symbolic state associated to each transition down to an abstract filter for the entire query and broadcast it back to every mapper, iv) filter the input events based on the abstract filter, and v) perform the reduction step over the remaining events. For both the baseline and our approach, the final reduction step just collects the events from the mappers, but does not perform the pattern matching. We made this choice in order to asses our solution independent from a particular implementation of the pattern matching operator and to underscore the fact that any of the existing complex event processing systems could benefit from our approach. Moreover, as the cost of pattern matching is typically proportional to the size of the input, taking it into consideration would more negatively impact the baseline approach than ours.

By discarding some of the input events our solution cuts the cost of the sorting and reduction phases when compared to the baseline approach, while adding the overhead of building and applying the abstract filters. When the number of events removed is significant this leads to an overall lower processing time as can be seen in figure 7. Since every join predicate included in the abstract filter incurs additional costs when building and applying the filter, to maximize performance one should consider only the smallest/cheapest set of predicates able to filter the input down to the order of gigabytes/tens of gigabytes.

In particular the additional reduction in data provided by the Time filter for query A5 does not translate in lower processing time (see figure 7a). This happens because the default MainKey filter already drastically reduces the amount of data that needs to be processed by the pattern matcher. Therefore the performance gain from further reduction is easily canceled by the cost of building the additional filter.

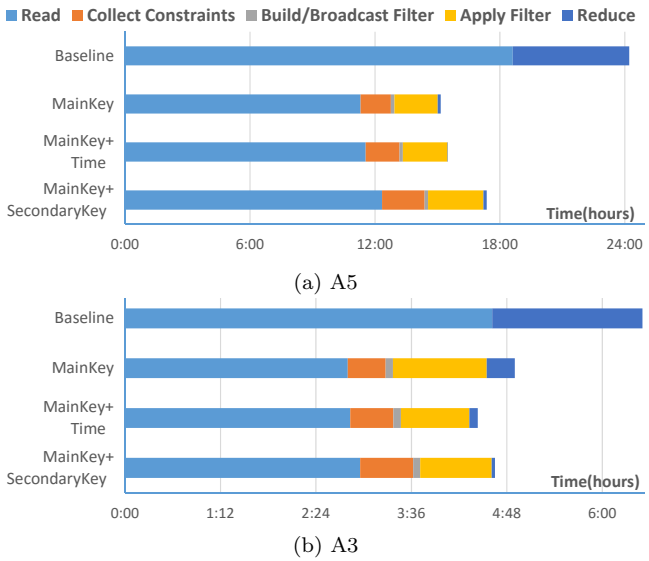


Figure 7: Breakdown of the total processing time for queries A5 and A3.

A similar situation occurs in query A3 where the sweet-spot is provided by the combination of MainKey and Time predicates, even though the combination of Main and Secondary Key filters manages to discard the largest number of events (figure 7b).

6.4 Latency speedup

In terms of end-to-end running times we record speedups between 1.08x and 1.78x for 8 out of the 18 queries in our workload, while for 4 of them our approach performs within 5% of the baseline (see Figure 5c). Just like in the case of processing times, the slowdowns in latency mainly correspond to queries for whom the abstract filters do not significantly reduce the amount of data processed by the pattern matcher (below 5x).

We examine the performance of queries A12 and A13 as it highlights another important factor in the latency speedup achieved by our approach. While for both queries the abstract filters discard a significant number of the input events and as such have smaller total processing times than the baseline, only for A12 this translates in smaller end-to-end running times. This happens due to the difference in the initial size of the input, 1.2TB for A12 vs 307GB for A13, which when processed on a 85 nodes cluster results in average running times of the reduction phase of 3.8 minutes for A12 vs 46 seconds for A13. Even though in our approach the reduction phase for A13 takes only 5.6 seconds on average due to the smaller number of events being processed, that is not enough to compensate for the additional latency introduced by the phases that build and broadcast the abstract filters. While the latency of these phases can be minimized by decentralizing the process of building the abstract filters, developing and deploying such techniques falls outside the scope of our current work.

7. CONCLUSIONS AND FUTURE WORK

Complex event processing and pattern matching is becoming commonplace in many analytics workloads over increasingly larger datasets. Consequently, it has to be able

to cope with a growing number of constraints wrt. to the physical layout of the data as well as enable optimization opportunities at the level of the entire processing pipeline. In our work, we translate a large class of commonly occurring patterns to relational queries in order to take advantage of decades of progress in relational optimizations. This is of particular importance, considering that most stages of data analytics workloads are also relational. In addition, we propose the technique of *abstract pattern matching* which leverages the relational representation of patterns to derive an abstract filter which discards those events guaranteed not to participate in a complete match and we explore the trade-off between the accuracy vs the operating cost of the filter by designing set abstractions that efficiently represent the domain of the join attributes in the pattern. Finally, we show that *abstract pattern matching* is effective in dramatically reducing the amount of data that needs to be shuffled over the network and processed by the pattern matching operator (from terabytes to gigabytes), and thus provide significant speedups to the pattern mining task.

While we can currently translate to relational queries the vast majority of complex event patterns encountered in the literature and in an industrial benchmark, we would also like to formally define the largest class of patterns for which such a translation is possible.

The approach we took in designing *abstract pattern matching* can be extended to optimize a large class of user-defined aggregates (UDA) that face challenges wrt. to the physical layout of their input data similar to those of pattern matching operators. To do so, one has to first collect symbolic sets for the variables binding input tuples, and then refine them by propagating constraints while interpreting the UDA on top of them. The symbolic sets that result from following paths that lead to successful outputs can then be used to discard irrelevant tuples from the input. Abstraction, both in terms of UDA's interpretation as well as the representation of sets, is bound to play a key role, just like it did in our refinement from precise to abstract filters.

8. REFERENCES

[1] Aster nPath Guide. https://developer.teradata.com/aster/articles/aster-npath-guide.

[2] Esper EPL Reference: Patterns. http://www.espertech.com/esper/release-5.1.0/esper-reference/html/event_patterns.html, 2016.

[3] FlinkCEP - Complex event processing for Flink. https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/libs/cep.html, 2016.

[4] Pattern Recognition With MATCH RECOGNIZE. https://docs.oracle.com/cd/E28280_01/apirefs.1111/e12048/pattern_recog.htm#CQLLR1531, 2016.

[5] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 147–160, 2008.

[6] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, Aug. 2008.

[7] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 3–14, 2013.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[9] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: A high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1100–1102, 2007.

[10] L. Brenna, J. Gehrke, M. Hong, and D. Johansen. Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 3. ACM, 2009.

[11] B. Cadonna, J. Gamper, and M. H. Böhlen. Sequenced event set pattern matching. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 33–44, 2011.

[12] B. Cadonna, J. Gamper, and M. H. Böhlen. Efficient event pattern matching with match windows. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 471–479, 2012.

[13] D. Carasso. *Exploring Splunk*. 2012.

[14] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[15] B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. *Proc. VLDB Endow.*, 3(1-2):220–231, Sept. 2010.

[16] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, 1977.

[17] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007*, pages 412–422, 2007.

[18] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, 1997.

[19] M. Hirzel. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 191–200, 2012.

[20] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Monitoring regular expressions on out-of-order streams. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE*, pages 1315–1319, 2007.

[21] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[22] A. K. Leghari, M. Wolf, and Y. Zhou. *Efficient Pattern Detection Over a Distributed Framework*, pages 133–149. 2015.

[23] M. Li, M. Liu, L. Ding, E. A. Rundensteiner, and M. Mani. Event stream processing with out-of-order data arrival. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, ICDCSW '07, 2007.

[24] V. Raychev, M. Musuvathi, and T. Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 153–167, 2015.

[25] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, June 2004.

[26] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *Proc. of the International Conference on Management of Data*, SIGMOD '14, pages 217–228, 2014.