Context-aware Event Stream Analytics

Olga Poppe[*], Chuan Lei[**], Elke A. Rundensteiner[*] and Dan Dougherty[*]

[*]Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609, USA

[**]NEC Labs America, 10080 N Wolfe Rd, Cupertino, CA 95014, USA

opoppe|rundenst|dd@cs.wpi.edu, chuan@nec-labs.com

ABSTRACT

Complex event processing is a popular technology for continuously monitoring high-volume event streams from health care to traffic management to detect complex compositions of events. These event compositions signify critical "application contexts" from hygiene violations to traffic accidents. Certain event queries are only appropriate in particular contexts. Yet state-of-the-art streaming engines tend to execute all event queries continuously regardless of the current application context. This wastes tremendous processing resources and thus leads to delayed reactions to critical situations. We have developed the first context-aware event processing solution, called CAESAR, which features the following key innovations. (1) The CAESAR model supports application contexts as first class citizens and associates appropriate event queries with them. (2) The CAESAR optimizer employs context-aware optimization strategies including context window push-down strategy and query workload sharing among overlapping contexts. (3) The CAESAR infrastructure allows for lightweight event query suspension and activation driven by context windows. Our experimental study utilizing both the Linear Road stream benchmark as well as real-world data sets demonstrates that the context-aware event stream analytics consistently outperforms the state-of-the-art strategies by factor of 8 on average.

1. INTRODUCTION

Complex Event Processing (CEP) has emerged as a prominent technology for supporting applications from financial fraud [30] to health care [32]. Traditionally, CEP systems consume event streams produced by smart digital devices like sensors and mobile phones and *continuously* evaluate the query workload to monitor the input event streams.

In many stream-based applications, events convey particular *application contexts* such that the system reaction to an event may significantly vary depending on the current context. Therefore, some event queries may only need to be executed under certain circumstances while others can be safely suspended. The following examples highlight the challenges and opportunities of context-aware event stream processing that have been overlooked in the prior research.

Motivating Example. Traffic has both a huge economic and environmental impact on our daily lives. Drivers traveling the 10-worst U.S. traffic corridors annually spend an average of 140 hours idling in traffic [2]. Due to pollution and noise, congestion in the USA's 83 largest urban areas in 2010 led to a related public health cost of \$18 billion [3]. Further, road traffic injuries caused an estimated 1.24 million deaths worldwide in 2010 [4].

An intelligent traffic control center could reduce these crippling impacts. The center receives vehicle position reports, analyzes them, infers the current situation in the monitored road segments and reacts instantaneously to ensure safe and smooth traffic flow. Early detection and prompt reaction to critical situations are eminently important. They prevent time and fuel waste, reduce pollution, avoid property damage and in some cases even save human lives.

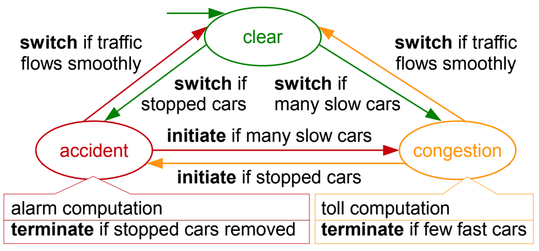

clear
switch if traffic flows smoothly
switch if stopped cars
switch if many slow cars
switch if traffic flows smoothly
accident
initiate if many slow cars
congestion
initiate if stopped cars
alarm computation
terminate if stopped cars removed
toll computation
terminate if few fast cars

Figure 1: CAESAR model of traffic management

System reaction to a position report should thus be modulated depending on the *current situation on the road* (here referred to as context[1]). Indeed, if an *accident* is detected, all vehicles downstream should be warned and possibly alternative routes should be suggested (Figure 1). If a road segment becomes *congested*, drivers may be charged toll to discourage them from driving to control smooth traffic flow. If a road segment is *clear*, none of the above actions should take place. Clearly, *current application contexts* must be rapidly detected and continuously maintained to determine appropriate reactions of the system at all times.

Conditions implying an *application context* can be complex. They are specified on both the event streams and the current contexts. For example, if over 50 cars per minute

[1]Here we utilize the term *application context* to refer to the state of a sub-network such as *accident*, *congestion*, etc. We deliberately avoid using the notion *state* since it is a too overloaded term in the CEP literature.

move with an average speed less then 40 mph and the current context is no *congestion* then the *context deriving query* updates the context to *congestion* for this road segment. To save resources and thus to ensure prompt system responsiveness, such complex context detection should happen once. Its results must be available on-time and shared among all queries that belong to the detected context. In other words, *context processing* queries are *dependent* on the results of *context deriving* queries and a mechanism ensuring their correct execution must be employed.

The system responsiveness can be substantially improved by exploiting the optimization opportunities enabled by the application contests. (1) Only those event queries that are relevant in the current contexts should be executed. All irrelevant computations should be suspended. (2) Workloads of overlapping contexts should be shared. Furthermore, application contexts break the application semantics into modules that facilitate the modular development and runtime maintenance of an event stream processing application.

Challenges. To enable such event stream processing applications, the following challenges must be tackled:

Context-aware specification model. As motivated above, event stream processing applications need to express rich semantics. In particular, they have to specify application contexts as first class citizens and enable linkage of appropriate event queries to their respective context. Furthermore, this model must be in a convenient human-readable format to facilitate on-the-fly reconfiguration, easy maintenance and avoid fatal specification mistakes.

Context-exploiting optimization techniques. To meet the demanding latency constraints of time-critical applications, this powerful context-aware application model must be translated into an efficient physical query plan. This query plan must be optimized by exploiting the optimization opportunities enabled by context-aware event stream analytics. This is complicated by the fact that the duration of a context is unknown at compile time and potentially unbounded.

Context-driven execution infrastructure. An efficient runtime execution infrastructure is required to support multiple concurrent contexts. To ensure correct query execution, the inter-dependencies between complex context deriving and context processing queries must be taken into account.

State-of-the-Art. The challenges described above have so far not been addressed in a comprehensive fashion.

Since the duration of a context varies, state-of-the-art window semantics such as fixed-length tumbling and sliding windows [22, 8] are inadequate to model the proposed notion of a context. Classical predicate windows [15] have variable duration. However, conditions leading to an application context can be rather complex and thus resource-consuming, worse yet they can be dependent on the previous contexts (Figure 1). Since predicate windows are independent from each other, they fail to express context windows.

While some event query languages (e.g., CQL [10], SASE [5, 34]) could be used to hard-code the equivalent of a context construct by queries that detect the context bounds. However, this approach is cumbersome and error-prone – requiring the careful specification of multiple complex interdependent event queries [25]. Furthermore, no optimization techniques have been developed to exploit the benefits of context-awareness such as suspension of irrelevant event queries nor the sharing workloads of overlapping contexts.

Business models [16, 28] focus on powerful modeling constructs to capture the semantics of processes and in that sense express application contexts. However, these models, targeting business process specification, were not designed for event stream processing. Thus, they neglect its core peculiarities such as the event-driven nature of context detection achieving high performance analytics and the importance of temporal windows and their processing techniques.

The Proposed CAESAR Approach. In [25], we formally defined the first context-aware event query processing model for which we now design the Context-Aware Event Stream Analytics in Real time system, CAESAR for short.

Our CAESAR model supports *context windows* as first-class citizens and associates appropriate event queries with each context window. Event queries that process events within a context are called *context processing queries.* Event queries that derive a context are called *context deriving queries.* Both types of queries operate within *context windows*, a new class of event query window we define.

To achieve near real-time system responsiveness, the CAESAR model is transformed into a stream query plan composed of context-aware operators of the *CAESAR algebra.* This algebra serves as foundation for the CAESAR optimizer. The optimizer exploits the notion of context windows to avoid unnecessary computations by suspending those operators which are irrelevant to the current context. Furthermore, the optimizer saves computations by sharing workloads of overlapping context windows. Finally, we built the *CAESAR runtime infrastructure* for correct yet efficient execution of inter-dependent context-aware event queries.

Contributions can be summarized as follows:

1) We introduce a new notion of windows, called *context windows*, to enable context-aware event query processing critical to modeling event-based systems. The proposed human-readable context-aware CAESAR model significantly simplifies the specification of rich event-driven application semantics by explicit support of context windows[2]. It also opens new multi-query optimization opportunities by associating appropriate event queries with each context.

2) We define the *CAESAR algebra* for our context-aware event query processing. The *CAESAR optimizer* pushes the context windows down to suspend the execution of irrelevant operators. Furthermore, we propose the context window grouping algorithm that exploits the sharing opportunities from workloads of overlapping context windows.

3) We built the *CAESAR runtime execution infrastructure* that guarantees correct and efficient execution of inter-dependent context deriving and context processing queries.

4) We evaluate the performance of the CAESAR system and its optimization strategies using the Linear Road stream benchmark [9] as well as the real world data set [26]. Our CAESAR system performs on average 8-fold faster than the context-independent solution for a wide range of cases.

Outline. We start with preliminaries in Section 2 and introduce the CAESAR model in Section 3. We present our algebraic execution paradigm in Section 4 and its optimization techniques in Section 5. Section 6 is devoted to the runtime execution infrastructure. We conduct the performance study in Section 7. Related work is discussed in Section 8, and Section 9 concludes the article.

[2]Visual editor for the CAESAR model and its evaluation, out of the scope of this article, are subjects for future research. In [25] we compare our model to a set of CQL event queries.

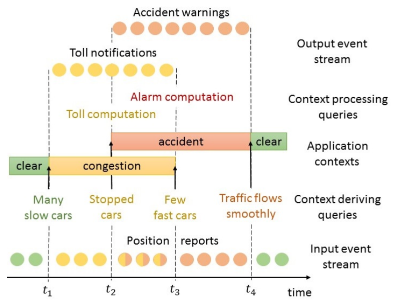


Figure 2: Key concepts of the CAESAR model

2. PRELIMINARIES

Time. Time is represented by a linearly ordered *set of time points* $(\mathbb{T}, \leq)$, where $\mathbb{T} \subseteq \mathbb{Q}^+$ and $\mathbb{Q}^+$ denotes the set of non-negative rational numbers. The *set of time intervals* is $\mathbb{TI} = \{[start, end] \mid start \in \mathbb{T},\ end \in \mathbb{T},\ start \leq end\}$. For a time point $t \in \mathbb{T}$ and an interval $w \in \mathbb{TI}$ we say that t is within w, denoted $t \sqsubseteq w$, if $w.start \leq t \leq w.end$.

Event. An *event* is a message indicating that something of interest happens in the real world. Each event e belongs to a particular *event type* E, denoted $e.type = E$. An event type E is defined by a *schema* which specifies the set of *event attributes* and the domains of their values. An event e has an *occurrence time* $e.time \in \mathbb{T}$ assigned by the event source. For example, a vehicle position report in [9] has the following attributes: expressway, direction, segment, car identifier etc. The values of these attributes are integers.

Event Stream. Events can be *simple* or *complex*. Simple events are sent by event producers (e.g., sensors) to event consumers (e.g., a traffic control center) on an input *event stream* I to be processed to derive higher-level complex events. The occurrence time of a complex event comprises the occurrence time of all events it was derived from [23].

3. CAESAR MODEL

3.1 Key Concepts of the CAESAR Model

Application contexts are real-world higher-order situations the duration of which is not known at their detection time and potentially unbounded. This differentiates contexts from events. The duration of a context is called a *context window*. For example, *congestion* is a higher-order situation in the traffic control application (Figure 2). Its bounds are detected based on position reports sent from cars in the same road segment in the same time period. As long as a road segment remains congested, the context window *congestion* is said to hold. Hence, the duration of a context window cannot be predetermined.

Context deriving queries associated with a particular context determine when this context should be terminated and when a particular other context is to be initiated based on events. For example, two context transitions are possible from the *congestion* context. If the number of cars reduces and they start moving at higher speed the system transitions into the *clear* context. If two cars stop in the same location and at the same time the *accident* context is activated.

Context processing queries correspond to the workload associated with a particular context, i.e., the analytics to be computed based on events received while the system remains in this context. For example, cars entering a congested road segment are charged toll to discourage drivers from driving during rush hours.

3.2 Benefits of Context-Awareness Property

Event Query Relevance. At each point of time, a context window re-targets all efforts of the system to the current situation by activating only those event queries (both context deriving and context processing queries) which belong to one of the currently active application contexts. All other event queries are suspended as they are irrelevant within the current contexts. This saves both CPU and memory resources. For example, toll is charged only during *congestion* on a road. This query is neither relevant in the *clear* nor in the *accident* contexts. Thus, it is evaluated only during *congestion* and suspended in all other contexts.

Event Query Simplification. The concept of an application context provides event queries with *situational knowledge* that allows us to specify simpler event queries. For example, if the event query computing toll is evaluated only during the *congestion* context, the complex conditions that determine that there is a traffic jam on the road are already implied by the context. Thus, there is no need to repeatedly double-check them in each of the active workload queries.

Context Derivation. The task of context derivation is the dedicated responsibility of the context deriving queries. For example, once too many slow cars in a road segment are detected, the context *congestion* is activated. Thereafter, the query detecting *congestion* is no longer evaluated. Also, all event queries that are evaluated during *congestion* leverage the insight detected by the context deriving query rather than re-evaluating the *congestion* condition over and over at each individual event query level.

3.3 Context Window

Definition 1. (**Context type** and **Context window**.) A context type is defined by a name c and a workload of context deriving queries Q_d^c and context processing queries Q_p^c which are appropriate in this context.

Let C be the set of context types. Then, a context window w_c is defined by a type $c \in C$ and a duration $(t_i, t_t] \in \mathbb{TI}$ where t_i is the time point when a query $q_i \in Q_d^{c'}$ matched the event stream and thus w_c got initiated and t_t is the time point when a query $q_t \in Q_d^c$ matched the event stream and thus w_c got terminated where $c' \in C$.

Context windows of different types may overlap. Indeed, there can be a *congestion* and an *accident* in the same road segment at the same time such that two sets of event queries handling both situations must be executed concurrently.

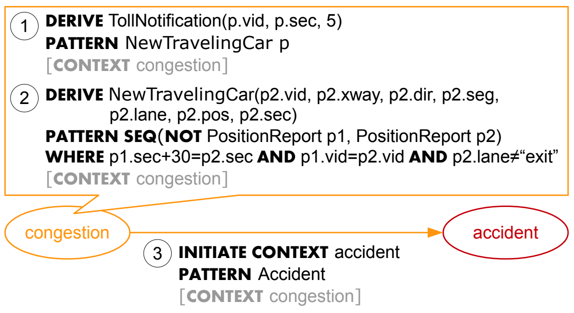
Definition 2. (**Context window relationships.**) Context windows of type c_1 and c_2 are guaranteed to overlap if based on the predicates of the respective context deriving queries it can be determined that for each window of type c_1 there is a window of type c_2 with $w_{c_1}.start \sqsubseteq w_{c_2}$. If in addition $w_{c_1}.end \sqsubseteq w_{c_2}$ can be determined, a window of type c_1 is contained in a window of type c_2.

In general, the predicates of the context deriving queries can be analyzed to determine if they imply such conditions. For example, Figure 7 shows the predicates that determine the bounds of the context windows w_{c_1} and w_{c_2}. It is easy to conclude that the windows overlap. CAESAR employs established approaches for predicate subsumption [14].

The same event query can be appropriate in several different application contexts. For example, accident detection happens in both the *clear* and the *congestion* contexts. In contrast to that, the event query detecting accident clearance is executed only in the *accident* context.

For simplicity, we have made two assumptions: (1) Event queries associated with different contexts are independent, meaning that they do not produce events that are consumed by event queries in other contexts. (2) Only one context window of the same type can hold at a time per road segment. If there are multiple accidents in a road segment the context window *accident* holds until all of them are cleared.

3.4 Context-aware Event Queries



```
DERIVE TollNotification(p.vid, p.sec, 5)
PATTERN NewTravelingCar p
[CONTEXT congestion]

DERIVE NewTravelingCar(p2.vid, p2.xway, p2.dir, p2.seg,
       p2.lane, p2.pos, p2.sec)
PATTERN SEQ(NOT PositionReport p1, PositionReport p2)
WHERE p1.sec+30=p2.sec AND p1.vid=p2.vid AND p2.lane≠"exit"
[CONTEXT congestion]

INITIATE CONTEXT accident
PATTERN Accident
[CONTEXT congestion]
```

Figure 3: Context-aware event queries

The two application contexts, *congestion* and *accident*, are shown in Figure 3. Different event queries are appropriate within them. For compactness, only three of them within the *congestion* context are shown. Clauses in square brackets are optional since they are implied by the model. The CAESAR event query language grammar is defined in Figure 4.

Definition 3. (**Context-aware event queries.**)
A context-aware event query consists of several clauses. Each clause performs one of the following tasks:
- Context initiation (INITIATE CONTEXT clause).
- Context switch (SWITCH CONTEXT clause).
- Context termination (TERMINATE CONTEXT clause).
- Complex event derivation (DERIVE clause).
- Event pattern matching (PATTERN clause).
- Event filtering (WHERE clause).
- Context window specification (CONTEXT clause).

Context deriving queries perform three actions: (1) initiate a new context window w_c, (2) terminate an existing context window w_c, or (3) switch from the current context window w_{c_1} into a new context window w_{c_2}.

Context initiation and termination can be used to express overlapping context windows. For example, *accident* and *congestion* may overlap. That is, query 3 initiates the context window *accident* when an accident is detected (Figure 3). However, query 3 does not terminate the context window *congestion*. The event queries that detect accidents are not shown for compactness.

In contrast, context switch expresses a sequence of two non-overlapping context windows. It corresponds to the termination of the previous context window w_{c_1} and the initiation of the new context window w_{c_2}. For example, the *clear* context overlaps neither *accident* nor *congestion* contexts.

Context processing queries analyze the stream of simple or complex events to derive higher-level knowledge in form of complex events. For example, query 2 detects the cars entering a congested road segment. These are vehicles which are not on an exit lane and for which there is no previous position report from the same road segment within 30 seconds. Query 1 derives toll notifications for such vehicles.

Both context deriving and context processing queries consume events that arrive during the context windows that these queries are associated with. Hence, both types of queries utilize event pattern matching and event filtering clauses which are commonly used in event queries [34, 23]. Section 4.1 defines when these clauses match.

$$
\begin{array}{lcl}
Query & := & \langle Window \rangle \mid \langle Retrieval \rangle \\
Window & := & (\mathsf{INITIATE} \mid \mathsf{SWITCH} \mid \mathsf{TERMINATE}) \\
 & & \mathsf{CONTEXT}\ Context \\
Retrieval & := & \langle Derive \rangle\ \langle Pattern \rangle\ \langle Where \rangle?\ \langle Context \rangle \\
Derive & := & \mathsf{DERIVE}\ EventType\ ((Var.)?\ Attr,?)+ \\
Pattern & := & \mathsf{PATTERN}\ \langle Patt \rangle \\
Where & := & \mathsf{WHERE}\ \langle Expr \rangle \\
Context & := & \mathsf{CONTEXT}\ (Context\ ,?)+ \\
Patt & := & \mathsf{NOT}?\ EventType\ Var? \mid \mathsf{SEQ}(\ (\langle Patt \rangle\ ,?)+\ ) \\
Expr & := & Constant \mid Attr \mid \langle Expr \rangle\ \langle Op \rangle\ \langle Expr \rangle \\
Op & := & + \mid - \mid / \mid * \mid \% \mid = \mid \neq \mid > \mid \geq \mid < \mid \leq \mid \mathsf{AND} \mid \mathsf{OR}
\end{array}
$$

Figure 4: CAESAR event query language grammar

Putting the application contexts, transitions between them (Definitions 1 and 2) and context-aware event queries (Definition 3) together, we now define the CAESAR model.

Definition 4. (**CAESAR model.**) A CAESAR model is a tuple (I, O, C, c_d) where I and O are unbounded input and output event streams and C is a finite set of context types with the default context type $c_d \in C$.

While the goal of classical automata is to define a language, the CAESAR model is designed for context-aware event query execution. Thus, final contexts are omitted. The CAESAR model has a default context that holds when no other context does, e.g., at the system startup (the *clear* context in our example). The runtime processing of the model is defined in Section 4.1.

4. CAESAR ALGEBRA

The CAESAR model explicitly supports application contexts and the transition network to facilitate context-aware event query specification (Figure 3). However, at execution level an algebraic query plan tends to be easier to optimize than an automaton-based model [30].[3] We thus define the CAESAR algebra and the translation rules of the CAESAR model into an algebraic query plan.

4.1 CAESAR Operators

The CAESAR algebra consists of six operators. While event pattern, filter and projection are quite common for other stream algebras [30], [34], context initiation, termination and context window are unique operators of the CAESAR algebra. Context initiation and termination consume a stream I of events produced by other operators of the context deriving queries and the set of current context windows

[3]There are approaches to optimization and distribution of simpler automata than the CAESAR model however. We describe them in detail in Section 8.

W. They update the set of the current context windows and return the updated set.

Context initiation CI_c starts a new context window w_c, adds it to the set of current context windows and removes the default context window w_{c_d} from the set, if there.
$\mathsf{CI}_c(I, W) := \{W' \mid$ If $w_c \in W$ then $W' = W$. Otherwise $W' = W \cup w_c$ and if $w_{c_d} \in W$ then $W' = W'/w_{c_d}$ where $e \in I$ and $e.time = w_{c_d}.end = w_c.start\}$.

Context termination CT_c ends the context window w_c, removes it from the set of current context windows, if the set becomes empty adds the default context window w_{c_d} to it.
$\mathsf{CT}_c(I, W) := \{W' \mid$ If $|W| > 1$ then $W' = W/w_c$ else $W' = \{w_{c_d}\}$ where $e \in I$ and $e.time = w_c.end = w_{c_d}.start\}$.

Context window CW_c consumes an event stream I and the set of all current context windows W and returns the stream of events that occur during the current context window w_c.
$\mathsf{CW}_c(I, W) := \{e \mid w_c \in W,\ e \in I,\ e.time \sqsubseteq w_c\}$.

Filter FI_θ with a predicate θ consumes an event stream I and returns a stream composed of all events that satisfy θ.
$\mathsf{FI}_\theta(I) := \{e \mid e \in I,\ e$ satisfies $\theta\}$.

Projection $\mathsf{PR}_{\mathsf{A,E}}$ with a set of attributes A and an event type E consumes an event stream I, restricts each input event to the set of attributes A and returns the stream of these restricted events of type E.
$\mathsf{PR}_{\mathsf{A,E}}(I) := \{e \mid e.type = E,\ e' \in I,\ e.a = e'.a$ for $a \in A\}$.

Pattern P consumes an event stream I and constructs event sequences matched by the pattern P. For each event sequence, the operator outputs an event consisting of the attribute values of all events in the sequence. Let A be the set of all attributes of events of types $E_1, ..., E_n$ and $1 \le i \le n$. The pattern P is one of the following:

1) Event matching E returns input events of type E.
$E(I) := \{e \mid e \in I,\ e.type = E\}$.

2) Sequence without negation $\mathsf{SEQ}_{(E_1,...E_n)}$ constructs sequences of n events such that an i^{th} event is of type E_i.
$\mathsf{SEQ}_{(E_1,...E_n)}(I) := \{e \mid e_1, ..., e_n \in I,\ e_1.type = E_1, ..., e_n.type = E_n,\ e_1.time < ... < e_n.time,\ e.a = e_i.a$ for $a \in A,\ e.time = [e_1.time, e_n.time]\}$.

3) Sequence with negation $\mathsf{SEQ}_{(S_1,\mathsf{NOT}\ E,S_2)}$ constructs event sequences SEQ_{S_1,S_2} without negation such that there is no event of type E between the sub-sequences constructed by SEQ_{S_1} and SEQ_{S_2}.
$\mathsf{SEQ}_{(S_1,\mathsf{NOT}\ E,S_2)}(I) := \{e \mid e_1, ..., e_m \in \mathsf{SEQ}_{S_1},\ e_{m+1}, ..., e_n \in \mathsf{SEQ}_{S_2},\ \nexists e' \in I$ with $e'.type = E,\ e_m.time < e'.time < e_{m+1}.time,\ e.a = e_i.a$ for $a \in A,\ e.time = [e_1.time, e_n.time]\}$.
A negated event can start or end an event sequence. In this case, temporal constraints must define the time interval within which the negated event may not occur [34].

4.2 Context-preserving Plan Generation

At compile time, the CAESAR model (Figure 3) is translated into an executable query plan that is than input to the CAESAR optimizer (Section 5). The CAESAR model translation happens in two phases (Figure 5). They are:

Phase 1: **CAESAR model to a query set.** First, the model is translated into a machine-readable query set. During this phase, contexts that are implied by the CAESAR model (the optional clauses in square brackets in Figure 3) become mandatory clauses of the CAESAR event queries. As a result, an event query that belongs to a context c has a mandatory clause CONTEXT c. For example, all queries in Figure 5 explicitly specify the context they belong to.

Phase 2: **Query set to a combined query plan.** During this phase, the machine-readable query set is translated into an executable query plan. This happens in two steps:
1) *Individual query plan construction.* Each event query is translated into a sequence of algebra operators such that each clause of the query corresponds to a set of operators as defined in Table 1. *curr* denotes the current context the context deriving query is associated with.
2) *Combined query plan construction.* Individual query plans are composed into a combined query plan such that if one query plan produces events which are consumed by another query plan then the output of the first plan is the input of the second plan. Since event queries in different contexts are independent (Section 3.3), all event queries in a combined query plan belong to the same context.

For example, queries 1 and 2 in Figure 3 are translated into the combined query plan in Figure 6(a). Query 1 is translated into the individual query plan consisting of operators 1–4. Query 2 corresponds to the individual query plan composed of operators 5–7. Since the first query plan produces complex events consumed by the second query plan, they are composed into a single combined query plan.

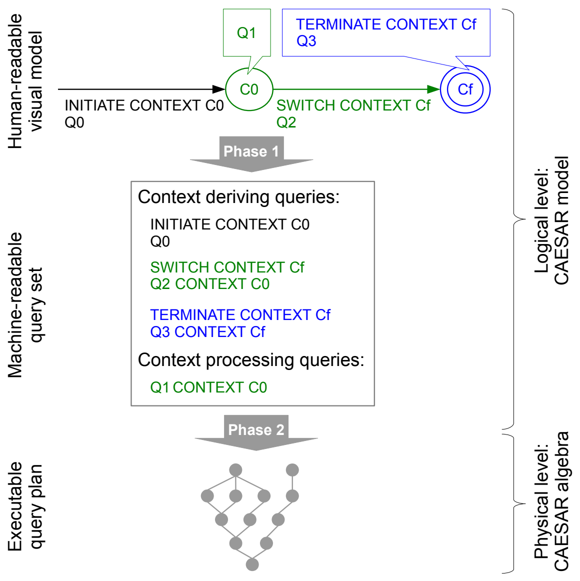


Figure 5: CAESAR model translation

Event query clause	Operator
INITIATE CONTEXT c	CI_c
SWITCH CONTEXT c	$\mathsf{CI}_c, \mathsf{CT}_{curr}$
TERMINATE CONTEXT c	CT_c
DERIVE $E(A)$	$\mathsf{PR}_{\mathsf{A,E}}$
PATTERN P	P
WHERE θ	FI_θ
CONTEXT c	CW_c

Table 1: Individual query plan construction

5. CAESER OPTIMIZATION

5.1 CAESAR Optimization Problem Statement

Definition 5. Given a workload of context-aware event queries where each query is associated with an application context. Our CAESAR optimization problem is to find an optimized query plan for all queries such that the CPU costs are minimized by suspending event queries that are irrelevant to the current application contexts and sharing the workload of overlapping context windows.

To avoid reinventing the wheel, we borrow the CPU cost estimation of event pattern construction from [24], and thus do not repeat here. Instead, we now discuss the cost of the context-specific operators. We maintain the information about current context windows in the *context bit vector W* with one bit for each context type. Since the number of possible context types for an application is predefined and constant, the size of the vector is also constant. The context initiation and termination operators update one bit and the time stamp of the context bit vector. Context windows look up one value in the vector to determine whether a context window of a certain type currently holds. In other words, the CPU cost of these operators is constant. Section 6 provides further implementation details.

5.2 Context Window Push Down

Since some operators of the CAESAR algebra are similar to other stream algebras, existing approaches, from operator reordering [24] to operator merging [30, 6], can be exploited by the CAESAR optimizer as well. For example, projections and filters can be executed in any order assuming that a projection that is pushed down below a filter discards no attributes accessed by the filter. Adjacent filters can be merged into a single filter by combining their predicates. However, these existing techniques are oblivious to the notion of contexts, and consequently do not avoid superfluous computations in the current contexts.

To avoid unnecessary computations when event queries are executed "out" of their respective context windows, we introduce the context window push-down strategy. Context window push-down can prevent the continuous execution of operators in the query plan. In other words, no event will be passed up by the context window operator if the current event stream does not qualify for the context window.

For instance, the two bottom most operators in Figure 6(a) are always executed regardless of the application contexts. Once the context window is pushed down to the bottom in Figure 6(b), it avoids the execution of all operators higher in the plan when they are irrelevant to the current contexts.

All event queries in a combined query plan belong to the same context (Section 4.2). By definition, a context window specifies the scope of its queries. Thus, pushing a context window down does not change the semantics of its queries. That is, context window push down strategy is correct.

Pushing a context window down seems similar to pushing a predicate or a traditional window down only at first sight. Differences are twofold:
1) Context windows *suspend* the entire query plan "above them" as long as the application is in different contexts. In contrast to that, a predicate or a traditional window is a filter on a stream that selects certain events to be passed through. It does not control the suspension of the above operators and keeps them in *busy waiting* state that wastes valuable resources and degrades system performance.
2) Our context-driven stream router directs *event stream portions during application contexts* to the appropriate event queries (Section 6.2). In contrast to that, predicates and traditional time constraints (e.g., event sequence within 4

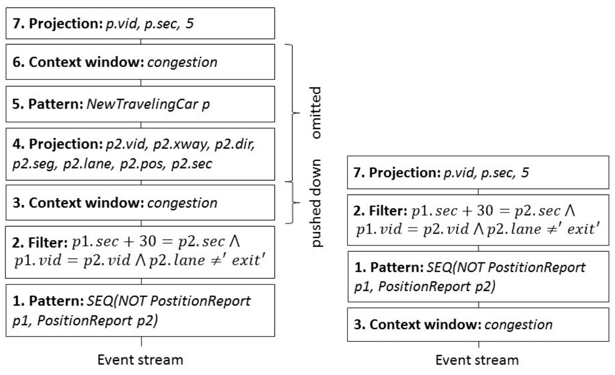

7. Projection: p.vid, p.sec, 5
6. Context window: congestion
5. Pattern: NewTravelingCar p
4. Projection: p2.vid, p2.xway, p2.dir, p2.seg, p2.lane, p2.pos, p2.sec
3. Context window: congestion
2. Filter: $p1.sec + 30 = p2.sec \wedge p1.vid = p2.vid \wedge p2.lane \neq' exit'$
1. Pattern: SEQ(NOT PostitionReport p1, PositionReport p2)
Event stream
omitted
pushed down
7. Projection: p.vid, p.sec, 5
2. Filter: $p1.sec + 30 = p2.sec \wedge p1.vid = p2.vid \wedge p2.lane \neq' exit'$
1. Pattern: SEQ(NOT PostitionReport p1, PositionReport p2)
3. Context window: congestion
Event stream
(a) Initial query plan (b) Optimized query plan

Figure 6: Query plans

hours [34]) typically filter events one by one at the *individual event level*. This is a resource consuming slow process.

Theorem 1 proves that pushing a context window down in a combined query plan leads to lower execution costs than placing it at any other position in the query plan.

THEOREM 1. *Given a query q, let P be the set of all possible query plans for q, $p \in P$ be a query plan for q and cost(p) be the cost of executing the plan p. With the context window pushed down, the new plan, denoted by p', has cost cost(p'). Then $\forall p \in P,\ p \neq p'.\ cost(p') \leq cost(p)$.*

PROOF. As described in Section 5.1, the cost of the context window operator is constant. That is, it adds constant cost to the overall execution costs of a query plan no matter its position in the query plan. The context window operator completely suspends the execution of all upstream operators while the context is not active. In that case, the cost of a query plan is reduced, i.e., $cost(p) < cost(p')$. In an unlikely case when the context window happens to be always active, the costs of the query plans p and p' are equal. □

5.3 Context Workload Sharing

Inspired by traditional multi-query optimization, grouping the event queries enables computation sharing among these queries. We observe the opportunity that substantial computational savings can be achieved by executing only one instance of each context deriving query for each context. Without context window grouping, each context processing query has to run its respective context deriving queries separately so to determine its current context to ensure correct execution. If this context analytics is performed on an individual query level, significant computational resources are wasted and system responsiveness suffers.

Sharing query workloads of overlapping context windows is challenging for the following reasons: (1) The duration of context windows may vary and their bounds are unknown at compile time. So, we have to infer whether context windows overlap. (2) Different contexts may contain identical or similar event queries in their query workloads. How to share query workloads driven by contexts is crucial to achieve high performance execution. We now propose an efficient solution that addresses this problem by splitting and grouping overlapping context windows.

For overlapping context windows, a naive solution would be to merge these context windows to form a larger encompassing context window. Inside this large context window, we can now analyze the associated event query workloads to optimize their executions. However, such solution could do more harm than good in some cases. For example, if all context windows were to be overlapping, then only one huge all encompassing context window would be formed as a result. This would forfeit the purpose of being context-aware. Consequently, redundant computations would be incurred.

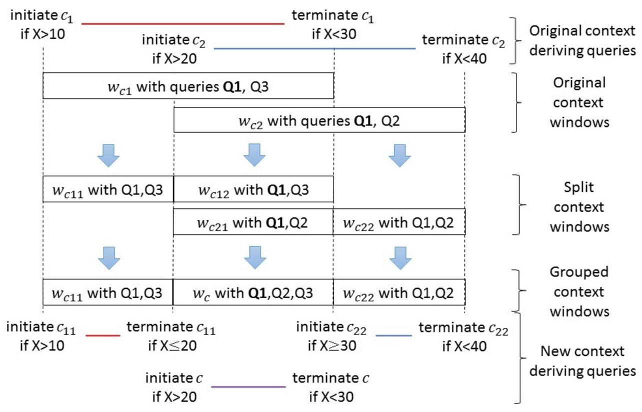


Figure 7: Context Window Grouping

We now propose an effective strategy for splitting the original user-defined *overlapping* context windows into finer granularity context windows and grouping them into *non-overlapping* context windows. For example, the context windows w_{c_1} and w_{c_2} in Figure 7 overlap. The window w_{c_1} is split into w_{11} and w_{12}, while the window w_{c_2} is split into w_{21} and w_{22}. Since the windows w_{12} and w_{21} cover the same time interval, the event queries associated with them are merged to form the workload of the *grouped context window* w. The context deriving queries are adjusted accordingly.

```
1   Input:  Set $W$ of user-defined context windows. A window is
            described by $start, end$ and $queries$.
3   Output:  Set $G$ of grouped context windows
    $G = W.extractNonOverlappingWindows()$
5   $W = W.sortByStart()$
    $W = W.mergeIdenticalWindows()$
7   $Q = \emptyset$
    while  $W.hasNextWindowBound()$
9         $next = W.getNextWindowBound()$
          $S = W.getStartingWindows(next)$
11        $E = W.getEndingWindows(next)$
          if  $Q.isEmpty()$
13        then  $Q = S.queries$
          else  new window $w = (previous, next, Q)$
15              $G = G \cup w$
                $Q = Q - E.queries \cup S.queries$
17        end if
          $previous = next$
19  end while
    for each  $w \in G$
21      $w.queries = w.queries.dropDuplicates()$
    end for each
23  return  $G$
```

Listing 1: Context window grouping algorithm

Consider our context window grouping algorithm in Listing 1. It takes a set of user-defined context windows as input. Context windows which do not overlap any other window remain unchanged (line 4). The algorithm sorts the overlapping context windows in increasing order by start time (line 5). Even though the exact start time of context windows is not known at compile time, the order of their beginning can be determined for overlapping context windows. For example, it is known at compile time that the window w_{c_2} in Figure 7 will start at the same time or later than the window w_{c_1}. If there are several identical context windows, the algorithm only keeps one by merging the workloads of identical windows (line 6). The core of the algorithm (lines 8-19) forms a new grouped context window for each time interval between two subsequent bounds of original context windows and associates the query workload with it that is appropriate during this time interval. For each grouped context window, the algorithm deletes duplicate event queries (lines 20-22) and returns the set of grouped context windows (line 23). Since several subsequent grouped context windows correspond to one original context window, an event query within a grouped context window may need access to its partial matches in the previous grouped context windows to ensure completeness of its results. In Section 6, we introduce a customized design (called *context history*) to ensure correct grouped context window execution.

The time complexity of the algorithm is $O(n\log(n) * m)$ where n is the number of original user-defined context types that are sorted (line 5) and m is the number of predicates that have to be analyzed while comparing two context types.

Based on the newly produced non-overlapping context windows, we now further exploit traditional multi-query optimization (MQO) techniques [31, 27, 21] to produce an optimized shared query execution plan for each group of event queries. This opens opportunities to share the similar workload within a context which further saves computational costs and reduces query latency. MQO is an NP-Hard problem due to the exponential search space. Thus, the solutions [31, 27, 21] of MQO tend to be expensive.

Our context window grouping solution divides the event query workloads into smaller groups based on their time overlap. Hence, the search space for an optimal query plan within each group is substantially reduced compared to the global space. Any state-of-the-art MQO solution can leverage this idea to return an optimized query plan efficiently.

The search space for multi-query optimization is doubly exponential in the size of the queries (n). The spectrum of possible multi-query groupings ranges from a separate group per each *individual* query (i.e., non-sharing) in the given query workload to a single group for *all queries*. The upper-bound for all possible multi-query groups corresponds to the number of distinct ways of assigning n event queries to one or more groups. The number that describes this value is the Bell number B_n, which represents the number of different groupings of a set of n elements. The Bell number is the sum of Stirling numbers. A Stirling number $S(n, k)$ is the number of ways to partition n elements into k partitions [20]:

$$B_n = \sum_{k=1}^{n} S(n,k) = \sum_{k=1}^{n} \left(\frac{1}{k!} \sum_{j=1}^{k} (-1)^{k-j} \binom{n}{k} j^n \right)$$

By dividing our n queries first into m groups, we subsequently only need to optimize the small shared set one by one and thus reduce the search space to:

$$B'_n = \sum_{k'=1}^{n/m} S(n/m, k') = \sum_{k'=1}^{n/m} \left(\frac{1}{k'!} \sum_{j=1}^{k'} (-1)^{k'-j} \binom{n/m}{k'} j^{n/m} \right)$$

As confirmed by our experimental study in Section 7.2, the CAESAR optimizer produces a context-aware query plan 2712 times faster than the state-of-the-art context-independent multi-query optimization approaches.

6. CAESAR EXECUTION INFRASTRUCTURE

6.1 Overview of the CAESAR Infrastructure

Figure 8 shows the CAESAR execution infrastructure. The boxes represent the system components.

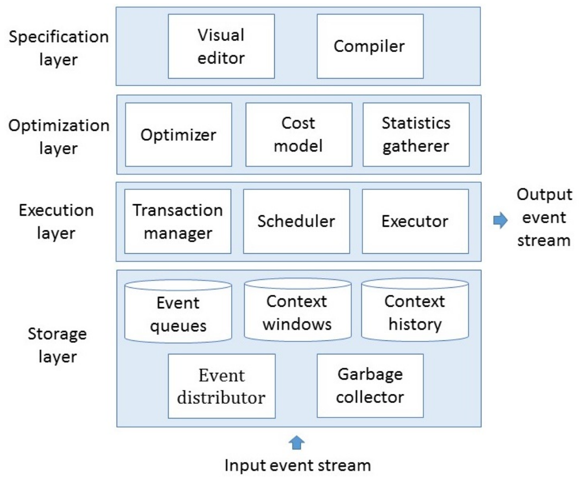


Figure 8: CAESAR infrastructure

Specification Layer. A CAESAR model is specified by the application designer using the visual CAESAR editor [25]. As explained in Section 4, we then translate it into an algebraic query plan.

Optimization Layer. As described in Section 5, the query plan is optimized using several context-aware optimization strategies to produce an execution plan.

Execution Layer. The optimized query plan is forwarded to the transaction manager that forms transactions. These transaction are submitted for execution by the scheduler that guarantees correctness. These components build the core of the CAESAR execution infrastructure. They are described in details in Section 6.2.

Storage Layer. The event distributor buffers the incoming events in the event queues. The current context windows and the context history are compactly maintained in-memory. The garbage collector ensures that only the values which are relevant to the current contexts are kept.

6.2 Core of the CAESAR Infrastructure

The core of the CAESAR execution infrastructure consists of the context derivation, context processing, context-aware stream routing and scheduling of these processes (Figure 9).

Context Derivation. For each stream partition (unidirectional road segment in the traffic management use case), we save which context windows currently hold in the context bit vector W. This vector W has a time stamp $W.time$ and a one-bit entry for each context type, i.e., $W.size = |C|$. The entries are sorted alphabetically by context names to allow for constant time access. The entry 1 (0) for a context c means that the context window w_c holds (does not hold) at the time $W.time$. Since context windows may overlap, multiple entries in the vector may be set to 1.

The context window vector is updated by the context deriving queries. Since *each* event in the stream can potentially

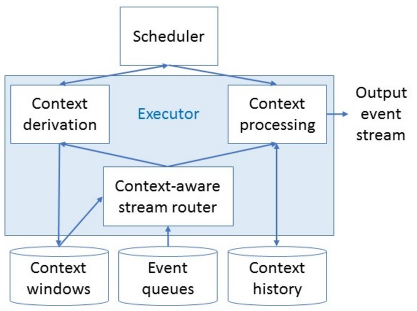


Figure 9: Core of the CAESAR infrastructure

update a context window, the context deriving queries processes all input events. $W.time$ is the application time when the vector W was last updated. Since events arrive in-order by time stamps, only one most recent version of the context bit vector is kept.

Context-aware Stream Routing. Based on the context window vector, the system is aware of the currently active event query workloads. For each current context window w_c, it routes all its events to the query plan associated with the context c. Query plans of all currently inactive context windows do not receive any input. They are suspended to avoid busy waiting, i.e., waste of resources.

Context-aware stream routing is a light-weight process for the following reasons. First, the lookup of all vector entries set to 1 takes constant time. Second, this routing happens for stream batches (multiple subsequent events in the input event stream) rather than for single events.

Context Processing. The CAESAR model allows the application designer to specify the *scope* of event queries in terms of their context windows. When a user-defined context window ends, all event queries associated with it are suspended and thus will not produce new matches until they become activated again. Therefore, their partial matches, called *context history*, can be safely discarded.

When a user-defined context window w_c with its associated query workload Q^c is split into smaller non-overlapping context windows w_{c_1} and w_{c_2} partial matches of the queries Q^c are maintained across these newly grouped windows w_{c_1} and w_{c_2} to ensure correctness of these queries Q^c. Therefore, for each event query we save the grouped context windows across which the results of the query are kept. For example, the event query q_1 in Figure 7 is executed during all 3 grouped context windows. However, when the third window begins, the partial results within the first window expire.

Correct Context Management. Context processing queries are dependent on the results of context deriving queries. Due to bursty input streams, network and processing delays context derivation might not happen on time. To avoid race conditions, these inter-dependencies must be taken into account to guarantee correct execution.

We define a *stream transaction* as a sequence of operations that are triggered by all input events with the same time stamp. The application time stamp of a transaction (and all its operations) coincides with the application time stamp of the triggering events. An algorithm for scheduling read and write operations on the shared context data is *correct* if conflicting operations[4] are processed sorted by time stamps.

[4]Two operations on the same value such that at least one of

While existing transaction schedulers can be deployed in the CAESAR system, we now describe a time-driven scheduler. For each time stamp t, our scheduler waits till the event distributor progress is larger than t and the context derivation for all transactions with time stamps smaller than t is completed. Then, the scheduler extracts all events with the time stamp t from the event queues, wraps their processing into transactions (one transaction per road segment for the traffic control use case) and submits them for execution.

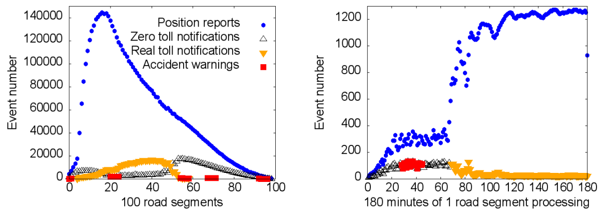
7. PERFORMANCE EVALUATION

7.1 Experimental Setup and Methodology

Experimental Infrastructure. We have implemented our CAESAR system in Java with JRE 1.7.0_25 running on Linux CentOS release 6.3 with 16-core 3.4GHz QEMU Virtual CPU and 48GB of RAM. We execute each experiment three times and report their average results here.

Linear Road Benchmark. We have chosen this benchmark [9] to evaluate the effectiveness of the CAESAR system for the following reasons: (1) it expresses a variety of application contexts such that the system reactions to an event depends on the current context, and (2) it is time critical since it poses tight latency constraint of 5 seconds.

Event Queries. We focused on a subset of event queries of the benchmark that based on input events derive toll notifications and accident warnings. The queries depicted in Figure 3 are simplified versions of the actual benchmark queries to illustrate the key concepts of our model in the paper only. We simulate low, average and high query workloads by replicating the event queries of the benchmark.



(a) Events per road segment (b) Events per minute

Figure 10: Event streams

Event Streams. Event distribution across road segments varies. Figure 10(a) shows the number of processed and derived events per segment of a randomly chosen unidirectional road[7]. There are more cars in some road segments than in others. In some road segments accidents and traffic jams happen more often than in others. Hence, more toll notifications and accident warnings are triggered for them.

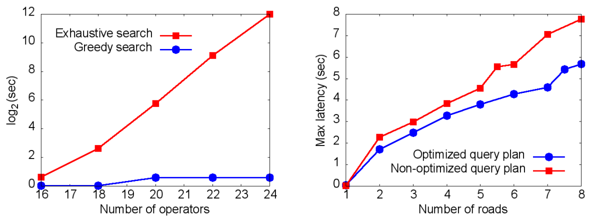
Event distribution across time also varies. Event rate gradually increases during 3 hours of an experiment. Figure 10(b) shows the number of processed and derived events per minute by for a randomly chosen unidirectional road segment[7] and visualizes the application contexts. Accident warnings are derived only during accidents (minutes 30-50). The benchmark requires zero toll derivation during accidents and clear road conditions (minutes 0-70). During traffic jams, real toll is computed (minutes 70-180).

Real Data Set. In addition to the benchmark, we evaluate our system using the physical activity monitoring real data set (1.6GB) [26]. It contains physical activity reports from 14 people during 1 hour 15 minutes.

Metrics. We measure two metrics common for stream systems, namely *maximal latency* and *scalability*. Additionally, we measure the *win ratio* of context-aware over context independent event stream analytics in terms of CPU processing time. *Maximal latency* is the maximal time interval elapsed from the event arrival time (i.e., system time when a position report is generated) till the complex event derivation time (i.e., system time when a toll notification or an accident warning is derived based on this position report). As mentioned above, the benchmark restricts the query latency to be within 5 seconds. The *system scalability* is determined by the *L-factor*, the maximal number of roads that are processed without violating this constraint. The *win ratio* of context-aware over context-independent event stream analytics is computed as the maximal latency of context-independent processing divided by the maximal latency of context-aware processing of the same event query workload against the same input event stream.

Methodology. To show the efficiency of our context-aware query optimization, we compare it to the exhaustive search approach by varying the number of operators in a query plan. To measure the effectiveness of our optimized context-aware query plan, we compare the performance of our context-aware query execution plan to the state-of-the-art solution [34, 5]. To demonstrate the effectiveness of our context-aware workload sharing technique, we compare it to our default non-sharing solution. We conduct these comparisons by varying the following parameters: number of operators in a query plan, input event stream rate, number of event queries, data distribution in a context window, length of a context window, number of context windows, number of overlapping context windows, overlapping ratio of context windows, and the shared workload size. Since context windows are derived from the input events, context window related parameters can be varied only through input data manipulation. Thus, for the experiments on real data set we vary the number of event queries.

7.2 Efficiency of CAESAR Optimizer



(a) CAESAR optimizer (b) L-factor

Figure 11: CAESAR Optimization Techniques

In Figure 11(a), we vary the number of operators in a query plan and measure the CPU time required for the query plan search (logarithmic scale on Y-axis). We compare the context-independent (CI) exhaustive to the context-aware (CA) greedy query plan search. As confirmed by the search space analysis (Section 5.3), the processing time of the exhaustive search grows exponentially with the number of operators in a query plan. In contrast, the CPU time required for our context-aware search stays fairly constant while varying the query plan size. At size 24, CAESAR's optimizer is 2712-fold faster than the exhaustive search. This is due to

them is a write are called conflicting operations.

[7]We have observed a similar event distribution for other roads and roads segments.

the fact that our context window push down and context grouping techniques substantially reduce the search space.

7.3 Efficiency of CAESAR Runtime

Next, we conduct a comprehensive evaluation to demonstrate the efficiency of our CAESAR solution. The baseline approach is the context-independent processing commonly used in most state-of-the-art solutions [34, 5, 32]. We first demonstrate the superiority of our CAESAR's context-aware event processing compared to the state-of-the-art context-independent approach by strictly following the constraints of the benchmark. Then we evaluate the CAESAR's context window sharing technique.

7.3.1 *Efficiency of Context-Aware Stream Analytics*

L-factor. In Figure 11(b), we vary the input stream rate by increasing the number of roads and measure the maximal latency. We compare the latency of the optimized versus non-optimized query plan. As the figure shows, the optimized query plan processes at most 7 roads without violating the latency constraint of 5 seconds. In contrast, the non-optimized query plan can process at most 5 roads under this constraint. Intuitively, our context-aware optimization approach successfully avoids the unnecessary computations by executing different queries only at appropriate time periods (contexts). On the other hand, the state-of-the-art solution suffers from executing all queries all the time.

Next, we compare continuous context-independent stream processing to context-aware solution when some of the involved event query workloads are appropriate only in certain critical contexts and can be suspended in other contexts. Unless stated otherwise, in Figures 12 and 13, we consider 3 roads (1.7GB) and assume that 2 critical non-overlapping context windows of length 3 minutes process 10 event queries each. These queries can be suspended in other contexts.

Evaluating diverse context window distributions. In Figure 13, we vary the number of event queries per context window and measure the maximal latency. We compare three setups, namely, uniform context window distribution versus their Poisson distribution with positive skew (λ is the first second) and with negative skew (λ is the last second). Context window bounds vary across different window distributions. The rest of the stream is identical for these setups.

As expected, when the context windows are at the end of the experiment where the stream rate is high, the maximal latency remains almost constant with the growing event query workload. This is explained by the fact that most queries are irrelevant for these contexts and thus are suspended. In contrast, if these windows are uniformly distributed or are at the beginning of the experiment when the stream rate is low, the maximal latency grows linearly with the number of queries. The maximal latency of 20 event queries with uniform context window distribution is 1.8-fold faster than with Poisson distribution with positive skew and 11-fold slower than with Poisson distribution with negative skew. Thus, to achieve fair results we consider uniform context window distribution in all following experiments.

Scaling event query workload. In Figure 12(a), we vary the number of event queries per context window and measure the maximal latency of context-aware versus context-independent event stream processing. The maximal latency grows linearly in the number of event queries. For an average workload of 10 event queries, we find that the context-aware processing is 8-fold faster than the context-independent solution using the Linear Road benchmark data (LR). CAESAR achieves the same win using the Physical Activity Monitoring data set (PAM) and 20 event queries. Our system has a clear win in this case because the context-aware event stream analytics suspends those event queries which are irrelevant to the current context.

Varying event stream rates. In Figure 12(b), we vary the number of considered roads. We measure the maximal latency of context-aware versus context-independent processing. The maximal latency grows linearly with an increasing input stream rate (number of roads). For 7 roads, context-aware processing is 9-fold faster than the context independent solution. The results show that the CAESAR system is more robust to the event stream rate increase compared to the context-independent solution.

Varying context window lengths. In Figure 12(c), we vary the length of the context windows and measure the win ratio of the context-aware over context-independent processing. The numbers above the bars indicate the percentage of the input event stream covered by the context windows that allow suspension of complex event query workload. Given that CAESAR only keeps one context active at a time, the win ratio exceeds 3 if such context windows cover more than 80% of the stream. It becomes negligible (almost 1) when they cover less than 50% of the input event stream.

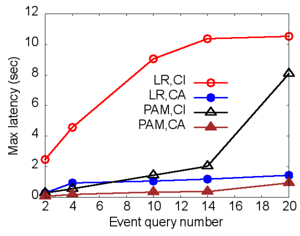
Varying the number of context windows. A similar trend can be observed while varying the number of context windows that allow suspension of irrelevant event queries (Figure 12(d)). Again, we measure the win ratio of the context-aware over context-independent stream processing. The numbers above the bars indicate the percentage of the input event stream covered by the context windows. Similarly, the win ratio exceeds 2 if the context windows cover more than 80% of the input event stream. It becomes negligible (almost 1) when they cover less than 50%.

7.3.2 *Efficiency of Context-Aware Workload Sharing*

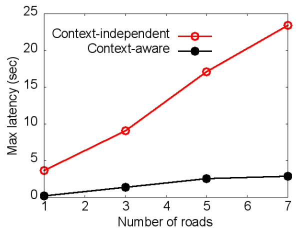
Next, we measure the effect of the shared workload processing of overlapping context windows (Figure 14). Unless stated otherwise, 30 windows of length 15 minutes each overlap by 10 minutes. Each of them processes 4 event queries.

Varying the number of overlapping context windows. In Figure 14(a), we vary the maximal number of overlapping context windows and measure the maximal latency of shared versus non-shared query processing. As expected, the larger the number of overlapping context windows are the more significant is the gain of the event query sharing. If 45 context windows overlap, the workload sharing strategy outperforms the default non-shared solution by factor of 10. The reason is that the CAESAR context window grouping technique exploits the sharing opportunities within overlapping context windows at a fine granularity level by splitting the overlapping context windows into non-overlapping parts and sharing event query processing within them.

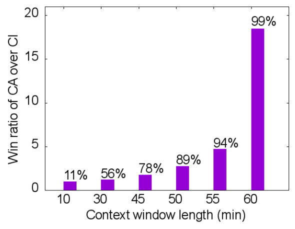
Varying the length of context window overlap. In Figure 14(b), we vary the minimal length of context window overlap and measure the maximal latency of shared versus non-shared workload execution. The gain of sharing grows linearly with the length of overlap. If 30 context windows overlap by 15 minutes, our workload sharing strategy performs 6-fold faster than the non-shared solution. This is due to the fact that similar workloads can be shared for a longer



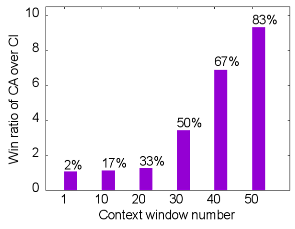
(a) Event query workload



(b) Event stream rate



(c) Context window length



(d) Context window number

Figure 12: Context-aware event stream analytics

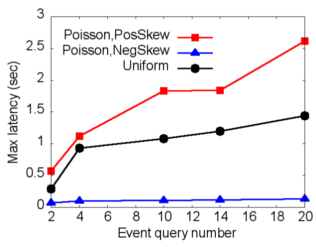
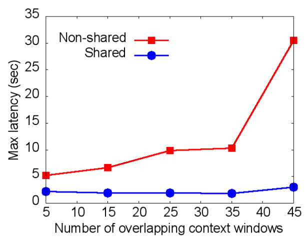
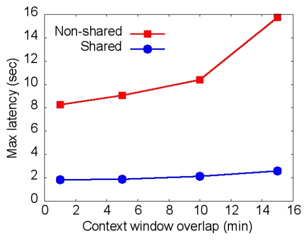


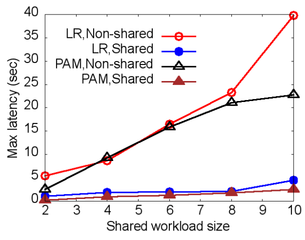
Figure 13: Context window distribution



(a) Degree of overlap



(b) Length of overlap



(c) Shared workload size

Figure 14: Shared workload of overlapping context windows

time period, and hence more computational savings can be harvested from the overlapping part of the context windows.

Shared workload size. In Figure 14(c), we vary the number of event queries per context window and measure the maximal latency of shared versus non-shared event query processing. As expected, the more event queries can be shared the more significant is the gain of the event query sharing. If each context window contains 10 queries that can be shared with other context windows, the workload sharing strategy outperforms the default non-shared solution by factor of 9 using the Linear Road benchmark data (LR). A similar trend can be observed with the Physical Activity Monitoring data set (PAM).

8. RELATED WORK

Context-aware Event Stream Models [11, 33] propose a fuzzy ontology to support uncertainty in event queries. Context-aware event queries are rewritten into context-independent and processed in parallel on different stream partitions. Isoyama et al. [18] propose to allocate event queries to event processors so that the state of event processing (i.e., intermediate event query results) is efficiently managed. These ideas are orthogonal to our optimization techniques.

Hermosillo et al. [17] and Proton software prototype [1] feature a model similar to the CAESAR model. However, these approaches lack a formal definition of the event language, optimization techniques and experimental evaluation. The recent emergence of these approaches confirms the importance of the notion of context-aware event queries that has been formally defined by us in [25]. Our current work on CAESAR now is completed by the context-aware optimization strategies and their experimental evaluation.

Event Stream Processing Automata [34, 5, 13] continuously evaluate the *same* set of *single* event queries using traditional windows of *fixed* length. These automata capture single query processing states. Their runs correspond to independent event query instances. In contrast to that, the CAESAR model expresses the semantics of the *whole stream-based application* rather than single isolated event queries. Our model captures application contexts of variable, statically unknown duration. Its runs correspond to interdependent processes traveling through application contexts, triggering appropriate context-aware event queries and incrementally maintaining their results.

Event Query Languages, like CQL [10] and SASE [5, 34], lack explicit support of application contexts. These contexts could possibly be hard-coded by queries deriving events that mark context bounds. However, this approach is cumbersome and error prone. It is neither modular nor human-readable. It requires tedious specification of multiple complex event queries – placing an unnecessary burden on the designer [25]. Inter-dependencies between context deriving and context processing queries would have to be taken into account to avoid re-computations, waste of valuable resources, delayed responsiveness and even incorrect results. Special optimization techniques would have to be developed to enable the benefits of context-awareness – as accomplished by our approach (Section 3.2). Furthermore, workload sharing among overlapping context windows has not been addressed in prior research.

Event Query Optimization Techniques [29] are often based on stream algebras [30, 34, 12]. Operators of these stream algebras work on *events*. In these approaches, *application contexts* are not supported as first-class citizens and thus they are not available for the operators. Hence, these approaches miss the event query optimization opportunities enabled by context-aware stream processing. Application contexts are first class citizens in our CAESAR model. They enable context-aware optimization techniques (Section 5).

Business Process Models [16, 28] explicitly support application contexts in a readable manner and allow to specify context-aware system reactions. However, these models target business process specification. They were not designed for CEP and neglect its peculiarities. In particular, the event-driven nature of streaming applications and the importance of temporal aspect are not given enough attention. Indeed, context transitions in these models are triggered by conditions and process flow rather than by events [19]. Temporal constraints are specified on clocks rather than on event time stamps [7]. These models do not derive higher level knowledge in form of complex events.

9. CONCLUSIONS

The responsiveness of time-critical decision-making event stream processing applications can be substantially speed-up by evaluating only those event queries which are relevant to the current situation. Inspired by this observation, we propose the first context-aware event stream processing solution, called CAESAR, which is composed of the following key components: (1) To allow for human-readable context-aware event query specification, we propose the CAESAR model that visually captures the application contexts and allows the designer to associate appropriate event queries with each context. (2) To achieve prompt system responsiveness, the model is translated into a query plan composed of the context-aware operators of the CAESAR algebra we propose. This algebra serves as a foundation for the CAESAR optimizer that suspends those event queries which are irrelevant to the current application context and detects workload sharing opportunities of overlapping contexts. (3) We built the CAESAR runtime execution infrastructure that guarantees correct and efficient execution of inter-dependent context deriving and context processing queries. The context-aware processing is shown to perform 8-fold faster on average than the context-independent solution when using the Linear Road stream benchmark and real world data sets.

ACKNOWLEDGEMENTS

This work was supported by NSF grants IIS 1018443 and IIS 1343620.

10. REFERENCES

[1] Proton. `https://github.com/ishkin/Proton`, 2015. [Online; accessed 10-December-2015].

[2] The Wall Street Journal. `http://www.wsj.com/articles/SB10001424052970203733504577024000381790904`, 2015. [Online; accessed 24-July-2015].

[3] USA Today. `http://usatoday30.usatoday.com/news/nation/2011-05-25-traffic-pollution-premature-deaths-emissions_n.htm`, 2015. [Online; accessed 24-July-2015].

[4] Wikipedia. `https://en.wikipedia.org/wiki/List_of_countries_by_traffic-related_death_rate`, 2015. [Online; accessed 24-July-2015].

[5] J. Agrawal et al. Efficient pattern matching over event streams. In *Proc. of Int. Conf. on Management of data*, SIGMOD '08, pages 147–160. ACM, 2008.

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

[7] R. Alur et al. Automata for modeling real-time systems. In *Proc. of Int. Colloquium on Automata, Languages and Programming*, ICALP'90, pages 322–335. Springer, 1990.

[8] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proc. of Int. Conf. on Very Large Data Bases - Volume 30*, VLDB '04, pages 336–347. VLDB Endowment, 2004.

[9] A. Arasu et al. Linear road: A stream data management benchmark. In *Proc. of Int. Conf. on Very Large Data Bases*, volume 30 of *VLDB'04*, pages 480–491. VLDB Endowment, 2004.

[10] A. Arasu et al. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[11] K. Cao et al. Context-aware distributed complex event processing method for event cloud in internet of things. *Journal of Advances in Information Science and Service Sciences*, 5(8):1212–1222, April 2013.

[12] S. Chakravarthy et al. Integrating stream and complex event processing. In *Stream Data Processing: A Quality of Service Perspective*, volume 36 of *Advances in Database Systems*, pages 187–214. Springer US, 2009.

[13] A. Demers et al. Cayuga: A general purpose event monitoring system. In *Proc. of Int. Conf. on Innovative Data Systems Research*, pages 411–422, 2007.

[14] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[15] T. M. Ghanem et al. Exploiting predicate-window semantics over data streams. *SIGMOD Rec.*, 35(1):3–8, Mar. 2006.

[16] A. Grosskopf et al. *The Process: Business Process Modeling using BPMN*. Meghan Kiffer Press, 2009.

[17] G. Hermosillo et al. Complex Event Processing for Context-Adaptive Business Processes. In *Belgium-Netherlands Software Evolution Seminar*, pages 19–24, Dec. 2009.

[18] K. Isoyama et al. A scalable complex event processing system and evaluations of its performance. In *Proc. of Int. Conf. on Distributed Event-Based Systems*, pages 123–126, New York, NY, USA, 2012. ACM.

[19] F. Joyce. *Programming Logic and Design, Comprehensive.* Thomson, 2008.

[20] M. Klazar. Bell numbers, their relatives, and algebraic differential equations. *J. Comb. Theory, Ser. A*, 102(1):63–87, 2003.

[21] W. Le et al. Scalable multi-query optimization for sparql. In *ICDE*, pages 666–677, 2012.

[22] J. Li et al. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, Mar. 2005.

[23] M. Liu et al. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *Proc. of Int. Conf. on Management of data*, SIGMOD'11, pages 889–900. ACM, 2011.

[24] Y. Mei and S. Madden. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In *Proc. of the SIGMOD Int. Conf. on Management of Data*, SIGMOD'09, pages 193–206. ACM, 2009.

[25] O. Poppe et al. The HIT model: Workflow-aware event stream monitoring. In A. Hameurlain et al., editor, *Advanced data stream management and continuous query processing*, volume 8 of *Transactions on large-scale data and knowledge-centered systems*, pages 26–50. Springer, 2013.

[26] A. Reiss et al. Creating and benchmarking a new dataset for physical activity monitoring. In *Proc. of Int. Conf. on PErvasive Technologies Related to Assistive Environments*, PETRA'12, pages 40:1–40:8. ACM, 2012.

[27] P. Roy et al. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD*, pages 249–260, 2000.

[28] N. Russell et al. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Proc. Asia-Pacific Conf. on Conceptual Modelling*, volume 53 of *APCCM*, pages 95–104. Australian Computer Society, Inc., 2006.

[29] S. Schneider et al. Tutorial: Stream Processing Optimizations. In *Proc. of Int. Conf. on Distributed Event-Based Systems*, DEBS'13, pages 249–258. ACM, 2013.

[30] N. P. Schultz-Møller et at. Distributed Complex Event Processing with query rewriting. In *Proc. of Int. Conf. on Distributed Event-Based Systems*, DEBS '09, pages 1–12. ACM, 2009.

[31] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[32] D. Wang, E. A. Rundensteiner, and R. T. Ellison, III. Active complex event processing over event streams. *Proc. VLDB Endow.*, 4(10):634–645, July 2011.

[33] Y. Wang et al. Context-aware complex event processing for event cloud in internet of things. In *Proc. of Int. Conf. on Wireless Communications and Signal Processing*, pages 1–6. IEEE, 2012.

[34] E. Wu et al. High-performance Complex Event Processing over streams. In *Proc. of Int. Conf. on Management of data*, SIGMOD '06, pages 407–418. ACM, 2006.