Temporal Order Optimizations of Incremental Joins for Composite Event Detection

François Bry
Institute for Informatics
University of Munich
http://www.pms.ifi.lmu.de/
francois.bry@ifi.lmu.de

Michael Eckert
Institute for Informatics
University of Munich
http://www.pms.ifi.lmu.de/
michael.eckert@pms.ifi.lmu.de

ABSTRACT

Queries for composite events typically involve the four complementary dimensions of event data, event composition, relationships between events (esp. temporal and causal), and accumulating events over time windows for negation and aggregation. We consider a datalog-like rule language for expressing such composite event queries and show that their evaluation can be understood as a problem of incrementally evaluating relational algebra expressions. We then show how temporal relationships between events can be utilized to make the evaluation of joins more efficient by avoiding evaluation of certain subexpressions and by making storage of some intermediate results unnecessary.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages

Keywords

Complex Event Processing, Composite Event Queries, Rules

1. INTRODUCTION

In distributed computer systems, events are omnipresent and exchanged as messages over networks. For many applications it is not sufficient anymore to query and react to only single, atomic events (i.e., events signified by a single message). Instead, events have to be considered with their relationship to other events in an event stream or cloud. Such events (or situations) that do not consist of one single atomic event but have to be inferred from some pattern of several events are called *composite* or *complex events*.

Amongst others, interest in them is driven by: a need to understand the dynamic behavior in distributed large-scale information systems [18]; increased generation of events from sensors due to drastic cost reductions in technology (e.g., RFID); a need to monitor log data generated in computer systems (e.g., fraud detection for credit cards); a need to monitor applications, services, and systems (e.g., business activity monitoring, monitoring of service level agreements); service-oriented architecture (e.g., accounting for on-demand services, synchronization of activities in business processes); and the emergence of event-driven architecture (e.g., detect and react to advantageous or dangerous situations) [12].

Expressing interest in certain composite events requires a (composite) event query language. A sufficiently expressive language should cover (at least) these four complementary dimensions:

Data extraction: Events contain data that is relevant for applications to decide whether and how to react to them. The data must be extracted and provided (typically as bindings for variables) to test conditions (e.g., arithmetic expressions) inside the query, construct new events, or trigger reactions (e.g., database updates).

Event composition: To support composite events, i.e., events that consist out of several events, event queries must support composition constructs such as conjunction and disjunction of events (more precisely, of event queries). Composition must be sensitive to event data, which is often used to correlate and filter events (e.g., consider only transactions from the *same* customer for composition). Since reactions to events are usually sensitive to timing and order, an important question is *when* a composite event is detected. In a well-designed language, it should be possible to recognize when events are detected and reactions triggered without difficulty.

Temporal (and causal) relationships: Time plays an important role in event-driven applications. Event queries must be able to express temporal conditions such as "events A and B happen within 1 hour, and A happens before B." For some applications, it is also interesting to look at causal relationships, e.g., to express queries such as "events A and B happen, and A has caused B." In this article we concentrate on temporal relationships. Causal relationships can be queried in essentially the same manner.[1] Note that according to Luckham's cause-time axiom [18], a causal relationship implies a temporal relationship; hence the temporal join optimizations discussed in this article apply also to causal relationships.

Event accumulation: Event queries must be able to accumulate events to support non-monotonic features such as negation of events (understood as their absence) or aggregation of data from multiple events over time. The reason for this is that the event stream is (in contrast to extensional data in a database) unbounded (or "infinite"); one therefore has to define a scope (e.g., a time interval) over which events are accumulated when aggregating data or querying the absence of events. Application examples where event accumulation is required are manifold. A business activity monitoring application might watch out for situations where "a customer's order has *not* been fulfilled within 2 days" (negation). A stock market application might require notification if "the *average* of the reported stock prices over the last hour raises by 5%" (aggregation).

[1] While temporality and causality can be treated similarly in queries, causality raises interesting questions about how causal relationships can be *defined* and *maintained*. Investigation of these issues is planned for the future.

DEBS '07, June 20-22, 2007 Toronto, Ontario, Canada

This work considers the evaluation of event queries formulated in a language that fully covers all four dimensions. To make the discussion concrete, we use a datalog-like rule language for querying events (Section 2). We show how such queries (more precisely, rule bodies) can be translated into relational algebra (Section 3). The problem of composite event detection (i.e., evaluation of composite event queries) can then be understood as a problem of incrementally evaluating relational algebra expressions that include temporal conditions on event occurrence times (Section 4). We then show how the temporal conditions can be utilized to make the evaluation more efficient by introducing two special (incremental) θ-joins (Section 5). The first θ-join makes storage of the intermediate results produced by one of its arguments unnecessary. The second θ-join can sometimes (depending events received so far) avoid evaluation of one of its arguments. We also explain that the two optimizations can be combined. Our approach is general enough to cover a wide range of composite event queries and we also discuss how our composite event detection approach relates to other approaches (Section 6).

The results in this paper are part of a work-in-progress on the high-level event query language XChange$^{\mathrm{EQ}}$ [7, 8], which supports events received as XML messages including SOAP [17] and CBE [11], and its efficient evaluation. The rule language we use here is a simplification that preserves the essential issues involved in event query evaluation. We focus on the evaluation of queries in rule bodies, chaining of rules is outside the scope of this paper.

To the best of our knowledge, it has not been made explicit in the literature so far that relational algebra can be used as a foundation for event query evaluation.[2] Explicitly using relation algebra for (composite) event detection has a number of advantages: It is simple and well-understood yet very expressive. Through equivalence laws it gives rise to query rewriting with the aim of obtaining more efficient query plans. It allows to build upon a myriad of research from the database community in particular for the implementation of operators. Of interest are in particular works on main memory databases (because event processing is usually done in main memory) as well as on distributed database (which gives rise to a distributed evaluation of event queries). Finally thinking in terms of relational algebra leads to optimized operators such as the novel temporal θ-joins introduced in this article.

2. QUERYING EVENTS WITH RULES

We now introduce a rule language for composite event queries. Events are seen as simple relational facts with an associated occurrence time. Following [14], events occur over time *intervals* rather then just at time points. It turns out that intervals are also very appropriate for (composite) events that have been derived by rules and thus should cover the full time interval of all their component events. For example the event $\mathrm{comp}(42, \text{"muffins"})^{[3,7]}$ indicates that an order for muffins with the number 42 has been completed over the time interval $[3, 7]$.[3]

Often events such as "order completed" are not present as an atomic event in the incoming event stream or cloud. Rather they have to be derived from existing events such as an event that a customer places an "order" with identifier id for a given quantity q of a product p followed by an event that this order (again with id) has been shipped with a tracking number t. This can be captured with the following rule:

$$(1)\quad \mathrm{comp}(id, p) \leftarrow o : \mathrm{order}(id, p, q), s : \mathrm{shipped}(id, t), o \text{ before } s$$

It illustrates the first three event querying dimensions. Two events of types "order" and "shipped" respectively are combined. Event data is extracted both to correlate the events on the order number id and for the construction of the new $\mathrm{comp(id, p)}$ event. Finally, a temporal condition specifies that the "order" event must happen before the "shipped" event (i.e., the end of the interval of "order" is less than start of the interval of "shipped" according to Allen's temporal relations for intervals [3]). Note that the "order" and "shipped" event have been given identifiers o and s respectively in the query that are then used to refer to them in the temporal condition. The occurrence time for the new "comp" event is the time interval covering all events detected in the body.

Events are received over time in an event stream which is assumed to extend indefinitely into the future. To be able to query with negation (in the sense of absence of events) one first has to restrict the infinite event stream to a finite extract. Once such a restriction (window or scope) is made, negation of events can be applied to the events accumulated in this window.[4] Accordingly we require an accumulation window to be specified in our language whenever negation is used. The accumulation window can be specified by another event (be reminded that events occur over time intervals not points) and is introduced by the keyword while.

Proceeding further with examples, let's say that an order is overdue if it hasn't been shipped within 6 hours in the case that less than 10 items were ordered and within 12 hours in the case of 10 or more items. Detecting "overdue" events involves a negation, i.e., the absence of "shipped" events in a given accumulation window (which is defined here by the event with identifier w):

$$(2)\quad \mathrm{overdue}(id) \leftarrow o : \mathrm{order}(id, p, q), w : \mathrm{extend}(o, 6h), \text{while } w : \text{ not } \mathrm{shipped}(id, t), q < 10$$

$$(3)\quad \mathrm{overdue}(id) \leftarrow o : \mathrm{order}(id, p, q), w : \mathrm{extend}(o, 12h), \text{while } w : \text{ not } \mathrm{shipped}(id, t), q \geq 10$$

Note that the event negation is (and must be!) sensitive to variable bindings. Only the absence of a "shipped" event with the same id as the "order" is relevant. In addition to the event accumulation required for negation, these rules show two more features: We can apply conditions on the event data such as $q < 10$ just like one can in any database query language. We can specify relative timer events, i.e., events whose occurrence time is defined relative to some other event. The event $\mathrm{extend}(o, 6h)$ begins with the start of an "order" event (recall that o its the event identifier) and ends 6 hours after the end of "order."

Such relative timer events are particularly useful when time-outs are involved (as in example above) or when sliding averages of values (or other aggregations) should be computed. Aggregation such as the computation of averages over data from several events is, like negation, a non-monotonic query construct and thus also requires the use of event accumulation.

As an example of a sliding average, consider reporting the number of all "shipped" events that have taken place in the last 24 hours whenever an "overdue" event is detected. A report of a high number could indicate that the shipping department is overloaded, a lower

[2] Of course relational algebra is a foundation of many data stream management systems. However, they usually use time only for windows in which data is collected; temporal relationships *between* events are usually not considered.

[3] For simplicity, we use integers to denote occurrence times here; of course our approach can deal with more realistic time models of human calendars as well.

[4] Keep in mind that accumulation here refers to the way we specify queries, not the way evaluation is actually performed. Keeping all events in the accumulation windows in memory is generally neither desirable nor necessary for query evaluation.

number that the problem is elsewhere.

$$(4)\quad \begin{array}{lll} \mathrm{rep}(\mathsf{count}(sid)) & \leftarrow & o : \mathrm{overdue}(oid), \\ & & w : \mathrm{extend\_backward}(o, 24h), \\ & & \mathsf{while}\ w :\ \mathsf{collect}\ \mathrm{shipped}(sid, t) \end{array}$$

This rule uses event accumulation (while) to collect all "shipped" events over a given time window. (Note that different variables oid and sid are used!) The time window is specified as a window going 24 hours into the past from the current "overdue" event ($\mathrm{extend\_backward(o, 24h)}$). The count aggregate function in the rule head is used to yield and report the desired number.

Note that this rule queries events that have been generated by other rules. The rationale for supporting deductive rules in an event query language is similar to that for views in databases: Rules serve as an abstraction mechanism, making query programs more readable. Rules allow to define higher-level application events from lower-level (e.g., database or network) events. Different rules can provide different perspectives (e.g., of end-user, system administrator, corporate management) on the same (event-driven) system. Rules allow to mediate between different schemas for event data. Additionally, rules can be beneficial when reasoning about causal relationships of events [18]. Semantics for (stratified) rule sets can be given as a model theory and fixpoint theory as is done in [8] for XChange$^{\mathrm{EQ}}$, which is an established approach for datalog [15].

In addition to deductive rules, event-based systems usually also require reactive rules, typically Event-Condition-Action (ECA) rules, to specify reactions to the occurrences of certain events. We do not address reactive rules here and refer to [8, 5] for a discussion of reactive rules and their differences to deductive (event) rules.

3. RELATIONAL ALGEBRA FOR EVENTS

To evaluate event query rules, we translate rule bodies into relational algebra expressions. These serve as a logical query plan and we can exploit query rewriting as an optimization technique. The actual incremental evaluation of (possibly rewritten) expressions will be the topic of later sections.

Whenever an event (e.g., $\mathrm{order}(42, \text{"muffins"}, 2)^{[3,3]}$) occurs that matches some atomic event query (e.g., $o : \mathrm{order(id, p, q)}$) this gives bindings for the free variables in the query (e.g., $id \mapsto 42, p \mapsto \text{"muffins"}, q \mapsto 2$) together with the event's occurrence time. We will represent the occurrence time as variable bindings with the special names $i.s$ and $i.e$, where i is the event identifier given in the query (e.g., $o.s \mapsto 3, o.e \mapsto 3$). This leads directly to representing the results of atomic event queries as relations of named tuples. Each atomic event query $\mathrm{i : Q}$ has an associated base relation R_i with schema $sch(R_i) = \{i.s, i.e\} \cup freevars(\mathrm{Q})$.

We can now translate composite event queries of rule bodies into relational algebra expressions in a straightforward manner. (Extended) projection is used to discard variables that do not occur in the rule head and to compute the occurrence time of the result. Combination of (atomic) event queries with conjunction is translated as a natural join. Conditions on the data are expressed as selections. Maybe a bit surprisingly, temporal conditions (such as o before s are also expressed as selections; this works because we made temporal information (i.e., occurrence times of events) part of the data of our base relations.

With this and R_o, S_s respectively denoting the relations for $o : \mathrm{order(id, p, q)}$ and $s : \mathrm{shipped(id, t)}$, the query from example (1) from the previous section can be expressed as:

$$\pi_{r.s \leftarrow \min\{o.s, s.s\}, r.e \leftarrow \max\{o.e, s.e\}, id, p}(\sigma_{o.e < s.s}(R_o \bowtie S_s))$$

The starting time $r.s$ of the result is the minimum of all involved starting times ($o.s$, $s.s$), the ending time $r.e$ is the maximum of all ending times ($o.e$, $s.e$).

Negation of events must be, as mentioned earlier, sensitive to variable bindings. It can be expressed using a θ-anti-semi-join, which is defined as $R \,\overline{\ltimes}_\theta\, S = R \setminus \pi_{sch(R)}(\sigma_\theta(R \bowtie S))$.

Relative timer events require the construction of their occurrence times and can thus be expressed by an extended projection.

The expression for example (2) then is (analogous for (3)):

$$\pi_{r.s \leftarrow \min\{o.s, w.s\}, r.e \leftarrow \max\{o.e, w.e\}, id}(\sigma_{q<10}(\\ \pi_{w.s \leftarrow o.s, w.e \leftarrow o.e+6h, sch(R_o)}(R_o) \,\overline{\ltimes}_{w.s<s.s \wedge s.e<w.e}\, S_s))$$

When event accumulation is used for aggregating data from events, this requires a θ-join between the accumulated events and the rest of the query, where the θ expresses the temporal condition given by the accumulation window. For the actual aggregation in the head, the grouping operator γ is used. (We follow the common notation and meaning for γ as given in [15]: its partitions the input tuples into groups of tuples having equal values on the grouping attributes and for each group outputs a single tuple with the grouping attributes and the additional aggregated attributed.)

With T_o and U_s denoting the relations for $o : \mathrm{overdue(oid)}$ and $s : \mathrm{shipped(sid, t)}$, the expression for example (4) is:

$$\gamma_{r.s, r.e, COUNT(sid)}(\pi_{r.s \leftarrow \min\{o.s, w.s\}, r.e \leftarrow \max\{o.e, w.e\}, sid}(\\ \pi_{w.s \leftarrow o.s-24h, w.e \leftarrow o.e, sch(T_o)}(T_o) \bowtie_{w.s<s.s \wedge s.e<w.e} U_s))$$

The framework laid out in this section is fairly general. Most event queries expressible in other event query languages, e.g., [1, 2, 4, 9, 10, 16, 19, 20, 22], can be translated into both the rule language and the relational algebra quite easily.[5] Further, using the same foundation as databases, our approach extends well to incorporate also non-event data from databases or other static data sources. This is useful to "enrich" event data in queries, e.g., events could have location identifiers and a database must be looked up to compute how close two events are in space to each other. Note that there typically is a semantic problem when database data changes during event detection, which we do not address in this work.

4. INCREMENTAL EVALUATION

Evaluating composite event queries, usually requires a data-driven, incremental approach for efficiency reasons: work done in one evaluation step (an evaluation step is performed whenever a relevant event occurs) of an event query should not be redone in future evaluation.

For example, when evaluating a rule like

$$(5)\quad \begin{array}{lll} \mathrm{d}(u, v, w, x) & \leftarrow & i : \mathrm{a}(u, v), j : \mathrm{b}(v, w), k : \mathrm{c}(w, x), \\ & & i \ \mathsf{before}\ k, j \ \mathsf{during}\ k \end{array}$$

any joins performed already between "a" and "b" events should not be recomputed upon reception of a "c" event.

The problem of evaluating of a composite event query Q_r (given as a relational algebra expression) can be formulated as a stepwise procedure as follows: In each evaluation step, we are given a set E of events (relational facts with associated occurrence time) that happened *at* the current time now (i.e., for all $e^{[t_1,t_2]} \in E : t_2 = now$). We are interested in all answers (composite events) produced by Q_r that happen *at* the current time now, i.e., we are required to deliver as a result only $Q_{new} = \sigma_{r.e=now}(Q)$. Note that computation of Q_{new} can however require knowledge of events that happened *before* the current time. This means that in addition to delivering the result we will have to also maintain some data structures that store old events for use in future evaluation steps.

[5] Note however that we do not discuss event consumption or instance selection [23] here.

We assume that the evaluation steps are performed in the order of event occurrence times here. A discussion of how this restriction can be lifted is given in Section 7.

Making the evaluation incremental concerns primarily the evaluation of joins. These are "blocking" operators, i.e., their current inputs may have to be combined with future inputs. For selection and projection this is not the case and they can directly output their results. (Note that the grouping operator as we used it in the previous section can be understood as non-blocking since the blocking has already been performed by a join in its input.)

The basic idea for making a join $R \bowtie S$ incremental is to have it store its inputs if they might be needed in future evaluation steps. We use the basic fact that $R \bowtie S = (R_{old} \bowtie S_{old}) \cup (R_{new} \bowtie S_{old}) \cup (R_{old} \bowtie S_{new}) \cup (R_{new} \bowtie S_{new})$, where R_{new} and S_{new} contain only the events happening at the current time, and R_{old} and S_{old} any (relevant) events that happened before. When performing the join in an evaluation step, $R_{old} \bowtie S_{old}$ need not be computed because it has already been computed by the previous steps.

The best way to describe such an incremental evaluation is to perceive each node in the expression tree as an object which has as children node objects for its subexpressions (arguments), some auxiliary data, and a method eval() which delivers the result of evaluating the expression (only those events happening at the current time). We additionally assume a global set E of all events happening at the current time as described above, which will be used for evaluating atomic events. The nodes for atomic event queries, selection, and joins can then be written in pseudo code as follows:

AtomicNode extends QueryNode:
  AtomicQuery A;
  Relation eval():
    return $A(E)$;

SelectionNode extends QueryNode:
  QueryNode Q;
  Condition C;
  Relation eval():
    return $\sigma_C(Q.\text{eval}())$;

JoinNode extends QueryNode:
  QueryNode Q_L, Q_R;
  Relation L_{old}, R_{old};
  Relation eval():
    L_{new} := Q_L.eval(); R_{new} := Q_R.eval();
    $J := (L_{new} \bowtie R_{old}) \cup (L_{old} \bowtie R_{new}) \cup (L_{new} \bowtie R_{new})$;
    $L_{old} := L_{old} \cup L_{new}$; $R_{old} := R_{old} \cup R_{new}$;
    return J;

Nodes for relative timer events (extended projections) potentially generate tuples with an occurrence time $j.e$ that lies in the future (i.e., $j.e > now$). These should not be passed on to the parent node immediately, but only in a later evaluation step. They are therefore stored in a relation $R_{delayed}$ until the time has progressed further:

RelativeTimerNode extends QueryNode:
  QueryNode Q;
  Relation $R_{delayed}$;
  EventAttribute i,j;
  Duration s', e';
  Relation eval():
    $R_{new} := \pi_{j.s \leftarrow i.s+s', j.e \leftarrow i.e+e', sch(Q)}(Q.\text{eval}())$;
    $J := \sigma_{j.e \le now}(R_{delayed} \cup R_{new})$
    $R_{delayed} := \sigma_{j.e > now}(R_{delayed} \cup R_{new})$
    return J;

Note that our approach does not assume that the relational algebra expressions have any special form. This means in particular that expressions obtained by the translation from the previous section can first be rewritten into more efficient logical query plans using the usual rules of relational algebra.

5. TEMPORAL JOIN OPTIMIZATION

When we look at how example (1) is evaluated, we can see that the join requires us to evaluate and store both inputs. By considering the temporal condition of the selection performed after the join, we could do better: It is unnecessary to store results of the "shipped" query — the temporal condition $o.e < s.s$ of the selection will discard any tuples generated by the joins $(R_{new} \bowtie S_{old})$ and $(R_{new} \bowtie S_{new})$. Further, it is unnecessary to evaluate the right input (the "shipped" query) as long as no "order" event has happened, again due to the temporal condition. Similar optimizations based on temporal conditions apply to the other examples.

We now introduce temporal θ-joins that allow to make such optimizations describing both the optimized evaluation of the θ-joins and the requirements on the condition θ. In general, θ will not be a single comparison (such as $o.e < s.s$) but a conjunction of multiple such comparisons. We will write $X < Y$ for the conjunction $\bigwedge_{x \in X} \bigwedge_{y \in Y} x < y$ for convenience. For ease of presentation we keep assuming that events arrive in temporal order.

5.1 Streaming Input

A θ-join node where the right input is streaming and not stored is a simple variation of a normal join node:

RightStreamingJoinNode extends QueryNode:
  QueryNode Q_L, Q_R;
  Condition θ;
  Relation L_{old};
  Relation eval():
    L_{new} := Q_L.eval(); R_{new} := Q_R.eval();
    $J := \sigma_\theta(L_{old} \bowtie R_{new})$;
    $L_{old} := L_{old} \cup L_{new}$;
    return J;

We have the following requirement: A join $Q_L \bowtie_\theta Q_R$ may be realized by a right streaming join node when the condition θ implies a comparison θ' of the form $\theta' = X < Y$ such that X is the set of all ending times occurring in the left subexpression Q_L and Y is a non-empty subset of the ending times occurring the right subexpression Q_R. Formally: $X = \{x.e \mid x.e \in sch(Q_L)\})$ and $Y \neq \emptyset, Y \subseteq \{y.e \mid y.e \in sch(Q_R)\})$.

To obtain the implied θ', the query rewriter usually has to use the fact that for any event i, $i.s \le i.e$. Accordingly, $\theta = o.e < s.s$ in the (rewritten) join $R_o \bowtie_{o.e<s.s} S_s$ for example (1) implies $\theta' = o.e < s.e$ and can thus be evaluated with a right streaming join.

It is important to see that in the general case a single comparison does not suffice for the condition θ'. We must really consider sets X, Y of time points on both sides of the comparison $X < Y$. An evaluation plan for example (5) could contain the following joins:

$$(R_i \bowtie S_j) \bowtie_{i.e<k.s \wedge k.s<j.s \wedge j.e<k.e} T_k.$$

The second join here can be evaluated with a right streaming join, because its condition implies $\{i.e, j.e\} < \{k.e\}$. However if we drop the condition j during k from the example rule (5), we would have $(R_i \bowtie S_j) \bowtie_{i.e<k.s} T_k$. A right streaming join would yield incorrect results here, because k events might have to be paired up with j events that happen later and thus must be stored. Of course the expression can be rewritten to the equivalent

$(R_i \bowtie_{i.e<k.s} T_k) \bowtie S_j$, where a right streaming join can be used for the first join.

The analogous optimization can be made for the left input and will be used in Section 5.3.

5.2 Suppressing Subexpression Evaluation

We now turn to a second optimization, where we avoid the evaluation of the right argument (subexpression) in a θ-join. Unlike the previous optimization, this optimization is dependent on the current state of the evaluation of the left input. We can only suppress evaluation of the right argument when we are sure that no input from the left will be paired up with an input that could come from the right side.

As knowledge about the current state of the evaluation, we need lower bounds on the starting and ending times of all events that might come (potentially in the future) as left input (denoted $lb(i.s)$, $lb(i.e)$, etc.). Note that the lower bound can in general not be computed just from the stored events L_{old} and the result of evaluating the left subexpression L_{new}. Consider again the expression $(R_i \bowtie S_j) \bowtie_{i.e<k.s} T_k$. The evaluation of T_k can be suppressed given $lb(i.e) \geq now$. However since i events are first joined with j events before arriving as input to the right join $\bowtie_{i.e<k.s}$, the lower bound of i cannot be derived just from the input to the right join.

Provided that event sources are able to deliver lower bounds,[6] we can simply compute the lower bounds as a map lb during the evaluation using that the evaluation is from left to right. Atomic nodes initialize $lb(i.s)$ and $lb(i.e)$, join nodes set $lb(t)$ to the minimum of the current $lb(t)$ and the lowest value for t in their stores L_{old} and R_{old} for each t. Nodes for relative timer events compute the lower bounds of their timer events from the current lb in the same manner as the corresponding extended projection. In the code from Section 4 on must therefore only add

$$lb(j.s) := lb(i.s) + s'; lb(j.e) := lb(i.e) + e';$$

before the return statement.

We can now give the pseudo code for a θ-join that suppresses evaluation of its right subexpression based on the value of lb for some time point attributes X (these will be defined below when we give the requirement on θ).

```
SuppressingJoinNode extends QueryNode:
    QueryNode QL, QR;
    Condition θ;
    TimePointAttributes X;
    Relation Lold, Rold;
    Relation eval():
        Lnew := QL.eval();
        for each t ∈ {y.s ∈ sch(QL)} ∪ {y.e ∈ sch(QL)}
            lb(t) := min{lb(t)} ∪ πt(Lold);
        if max{lb(x) | x ∈ X} ≥ now then
            J := ∅; R := ∅;
        else
            Rnew := QR.eval();
            J := σθ((Lnew ⋈ Rold) ∪ (Lold ⋈ Rnew)
                        ∪(Lnew ⋈ Rnew));
            Rold := Rold ∪ Rnew;
            for each t ∈ {y.s ∈ sch(QR)} ∪ {y.e ∈ sch(QR)}
                lb(t) := min{lb(t)} ∪ πt(Rold);
        Lold := Lold ∪ Lnew;
        return J;
```

[6]When the event source is another rule, this is possible without much difficulty. External event sources usually deliver only time point events and in this case the lower bounds are simply $lb(i.s) = lb(i.e) = now$. Relative timer events will be discussed shortly.

The requirement on the condition θ is that it implies a comparison $\theta' = X < Y$ such that Y is the set of all starting times occurring in the right subexpression Q_R and X is a non-empty subset of the starting and ending times occurring in the left subexpression Q_L. Formally: $X \neq \emptyset, X \subseteq \{x.s \mid x.s \in sch(Q_L)\} \cup \{x.e \mid x.e \in sch(Q_L)\}$ and $Y = \{y.s \mid y.s \in sch(Q_L)\}$. The set X is used in the pseudo code above. Note that we require a restriction on the *starting* times on the right side. If one of the event sources on the right side is a rule, this allows us to use a backward-chaining approach and suppress the evaluation of this rule. If the restriction is only on the ending time, such a rule must still be evaluated in order to update its event stores.

5.3 Combination

Both optimizations can be combined in some cases. The θ-join $R_o \bowtie_{o.e<s.s} S_s$ from example (1) can be evaluated both with the right input streaming and suppressing (when possible) evaluation of the right subexpression. This is a quite common case since temporal conditions of the form i before j are common in event queries.

There are also cases where we want the left input streaming while suppressing evaluation of the right subexpression. This is the case for the second join in

$$(R_i \bowtie S_j) \bowtie_{i.e<k.s \wedge k.s<j.s \wedge j.e<k.e} T_k.$$

from example (5). This is also a quite common case since it is introduced by temporal conditions of the form i during j as well as by event accumulation (form while $i : \ldots$).

Both combinations of the streaming and suppressing θ-joins and the necessary conditions are straightforward and not discussed here in depth for space reasons.

6. RELATED WORK

Popular approaches for the detection of composite events include finite state automata [16, 22, 4] and event trees (or graphs) [10, 19, 20, 1]. The automata approach explains well sequences of events occurring on time points. However it's unclear how it could be extended to events occurring over time intervals and to correlate data between events. The importance of time intervals for events is discussed in [14, 1]. The importance of correlating data between events has been emphasized in this article as well as in [2] and [9].

Our approach of translating event queries into relational algebra expressions that are evaluated incrementally can be understood as a generalization of event trees. The grounding in relational algebra explains better how event data is treated and gives a strong theoretical foundation, which opens the door to query optimization in the form of rewriting. As mentioned before, dealing with event data is also emphasized in [2] and [9], which are also based on the idea of event trees. The optimized θ-joins introduced in this article can be seen as generalizations some event tree operators such as sequence operators. There are two important differences though: First, rewriting has not been explored for event tree operators; this means that there is usually only one event tree (which corresponds to a query plan) for a given event query and, e.g., the sequence operator will *always* be used when the event query contains a sequence. This corresponds to choosing the joins order according to the temporal sequence of events. Our approach is more flexible, allowing for different join orders while still making temporal optimizations when the join order permits. This is favorable because the join order that makes temporal optimizations is not necessarily the most efficient (e.g., due to larger intermediate results).

The basic idea of incremental evaluation of joins can be found in the rete algorithm [13] as well as in work on view maintenance in databases [6, 21]. However, these works consider only facts which,

in contrast to events, do not have an occurrence time. Optimizations using temporal conditions, like those presented in this work, are therefore not considered there.

7. CONCLUSION AND FUTURE WORK

We have presented a method for detecting composite events based on an incremental evaluation of relational algebra expressions. Relational algebra provides a clear theoretical basis for reasoning about event queries, in particular query rewriting based on equivalence. Also evaluation of relational algebra (in particular joins) is very well investigated for distributed databases and this should give rise to considering distributed event query evaluation.

While we have made the assumption that events are processed in their temporal order, this assumption can be lifted, e.g., by assuming a maximal delay for events and buffering all events for the delay time. In the case of the (right) streaming join, the right input then requires a small buffer and event arriving out-of-order on the left have to be joined with all events in the right buffer. For the suppressing join, the set of current event E used when evaluating the right subexpression then becomes the set of all events in the buffer. This works well as long as no non-monotonic query constructs (aggregation and negation) are involved. In those cases the query processor must either wait for the full delay time to produce an answer or be allowed to alter given answers.

We have shown two optimizations for joins that rely on temporal conditions provided in the queries. We plan to investigate further (temporal) optimizations in the future, in particular relating to the removal of event which have "timed-out." Finally, the underlying idea of the suppressing join could be taken further: in this work, the right subexpression is either fully evaluated or not evaluated. Using variable bindings from the left input, it is also conceivable to evaluate the right subexpression only for the current variable bindings (again, provided that certain temporal conditions are satisfied). This would be similar to sideways information passing and applicable both to event sources that are other rules (via unification) and to external event sources which allow content-based subscriptions.

To apply such optimizations it is generally necessary to rewrite queries (based on the laws of relational algebra and (simple) reasoning about the temporal conditions). Development a query planner that uses rewriting rules to obtain such more efficient query plans is ongoing research. A major challenge here is that cost-based planning (as common in databases) is not readily applicable, because the common cost formulas do not consider a step-wise evaluation and the necessary estimates (cardinalities of relations, distribution of domain values) might not be available for event streams or clouds at the time the query is compiled.

Acknowledgments

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (`http://rewerse.net`).

8. REFERENCES

[1] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data and Knowledge Engineering*, 2005. In press.

[2] A. Adi and O. Etzion. Amit — the situation manager. *Int. J. on Very Large Data Bases*, 13(2), 2004.

[3] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11), 1983.

[4] R. S. Barga and H. Caituiro-Monge. Event correlation and pattern detection in CEDR. In *Proc. Int. Workshop Reactivity on the Web*, 2006.

[5] B. Berstel, P. Bonnard, F. Bry, M. Eckert, and P.-L. Pătrânjan. Reactive rules on the web. In *Reasoning Web, Int. Summer School*, 2007.

[6] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1986.

[7] F. Bry and M. Eckert. A high-level query language for events. In *Proc. Int. Workshop on Event-driven Architecture, Processing and Systems*, 2006.

[8] F. Bry and M. Eckert. Rule-Based Composite Event Queries: The Language XChange$^{\mathrm{EQ}}$ and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, 2007.

[9] F. Bry, M. Eckert, and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and applications of the language XChange. *J. of Web Engineering*, 5(1), 2006.

[10] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. Int. Conf. on Very Large Data Bases*, 1994.

[11] Common Base Event. www.ibm.com/developerworks/webservices/library/ws-cbe.

[12] O. Etzion. Towards an event-driven architecture: An infrastructure for event processing (position paper). In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, 2005.

[13] C. L. Forgy. A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intelligence*, 19(1), 1982.

[14] A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. Int. Conf. on Database and Expert Systems Applications*, 2002.

[15] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[16] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proc. Int. Conf. on Very Large Data Bases*, 1992.

[17] M. Gudgin et al. SOAP 1.2. W3C recomm., 2003.

[18] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[19] M. Mansouri-Samani and M. Sloman. GEM: A generalised event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2), 1997.

[20] D. Moreto and M. Endler. Evaluating composite events using shared trees. *IEE Proceedings — Software*, 148(1), 2001.

[21] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1996.

[22] C. Sánchez, M. Slanina, H. B. Sipma, and Z. Manna. Expressive completeness of an event-pattern reactive programming language. In *Proc. Int. Conf. on Formal Techniques for Networked and Distrib. Systems*, 2005.

[23] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proc. Int. Conf. on Data Engineering*, 1999.