On Static Determination of Temporal Relevance for Incremental Evaluation of Complex Event Queries

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

Evaluation of complex event queries over time involves storing information about those events that are relevant for, i.e., might contribute to, future answers. We call the period of time for which an event or an intermediate result must (at least) be stored its *temporal relevance*. This paper pioneers a precise definition of temporal relevance and develops a method for statically (i.e., at compile time) determining it. During query evaluation (i.e., at run time), this enables garbage collection of events that become irrelevant as time progresses. Temporal relevance is also important at compile time for cost-based query planning.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages; D.1.6 [**Programming Techniques**]: Logic Programming

Keywords

Complex Event Processing, Event Queries, Temporal Relevance, Garbage Collection, CERA, Events, Rules

1. INTRODUCTION

Complex event queries are standing queries that are evaluated over time against a stream of incoming (atomic) events, producing answers (complex events) on the fly and promptly. Their evaluation requires storing information about those received events that are relevant for, i.e., can contribute to, answers in the future. Consider an event query for an event of type A and another of type B to happen within 2 hours where both events have the same value for their single attribute x. The result should be provided as a complex event C having the single attribute x. In the event query and rule language of [8], this query is written as follows:

```
C(x) <- a: A(x), b: B(x), {a,b} within 2 hours
```

When during the evaluation of this query an event `A(42)` is received at a time t_1, then this event has to be stored since it might contribute to an answer to the query when an event `B(42)` is received at a later time $t_2 > t_1$.

While it is necessary to store received events for some time, it is for this query not necessary to store them forever. In our example, the condition that A and B happen within 2 hours of each other renders any A or B event that is older than 2 hours irrelevant for answers to this specific query or rule. Since we have only this single rule, any stored A or B event can be removed after 2 hours.

As we will see, determining temporal relevance becomes an involved problem when more complex queries than this example are considered and when we take into account an incremental evaluation.

Knowing how long an event is relevant is a prerequisite for performing garbage collection during event query evaluation, i.e., removing events that have become irrelevant from their stores and thus freeing up memory. It is also of central importance for developing cost-based query planners (akin to cost-based query planners found in traditional databases): an important input of any cost estimation function is the cardinality of event stores, which in turn is proportional to the length of time events are stored.

In addition to storing incoming events, many complex event query evaluation techniques also materialize and store intermediate results, which are sometimes also called semi-composed (complex) events since they incorporate (compose) information from several atomic events without yielding a full complex event (i.e., an answer to a query). The same problem of determining relevance applies to these intermediate results. We will see that the relevance of intermediate results is a noteworthy issue since it does not simply derive from the relevance of its constituent events.

This paper is the first to develop and define a clear notion of *temporal relevance* of events and intermediate results in the evaluation of complex event queries. Our objective is to derive *(temporal) relevance conditions*, that is, conditions for determining the relevance of events at query run-time.

We apply our ideas on query plans, which are based on a variant of relational algebra (called CERA here) and further have materialization points to account for (materialized) intermediate results in the incremental evaluation (Section 2).

After formalizing the problem (Section 3), we develop a method for statically (i.e., at compile time) determining temporal relevance conditions (Sections 4 and 5). When evaluated at query run time, they express whether a stored event tuple is still relevant (and must be kept), or whether it is irrelevant (and can be removed). These relevance conditions depend, of course, on the event queries, and we consider here the case where all queries are known in advance, i.e., before the actual query evaluation on the event stream starts. A generalization of our approach for different temporal conditions is discussed in Section 6.1. Solutions for dynamic situations, where queries can be added and removed dynamically at run-time, can be developed based on the static case as discussed in Section 6.2.

Our temporal relevance conditions will depend only on the time stamps in an event (or event tuple), giving a duration —called the relevance time— for which the event tuple has to be stored (and after which it can be deleted). The conditions will not require any involved operations such as joins with other events, which makes them fast to evaluate. Section 6.3 revisits this issue.

Our work is general so that it also covers those cases when an event query is not restricted in a way such that all events can always be removed after finite time, making it necessary for some events to be stored for a possibly infinite time. In particular it can serve as a tool for checking if a query can be evaluated with finite time bounds on event storage.

Temporal relevance is based only on time-related information such as occurrence times of events and temporal conditions in queries. We discuss less restricted forms of *(general) relevance* in Section 6.4. This includes, for example, *axiomatic relevance*, where relevance of an event is determined using axiomatic knowledge about the future event stream, i.e., which events can arrive in the future.

Determining relevance in complex event queries is an important issue with many facets beyond our concrete solution for temporal relevance (Section 6). Still, it is novel and little explored, due in part to restrictions of most event query languages (Section 7).

2. PRELIMINARIES

There is no commonly agreed theoretical foundation for event queries, yet. We base our work on the proposal in [8], which tries to leverage standard formalisms from databases for event queries. We now shortly reintroduce its cornerstones and extend it with a formal notion of "query plans with materialization points."

2.1 Rule-Based Event Query Language

To make discussions concrete, we express event queries using the rule language of [8] here. Note that this language is an abstraction of the high-level event query language XChange$^{\text{EQ}}$ [6, 7]. The main difference is that events are just relational facts here, while XChange$^{\text{EQ}}$ supports queries against events represented as XML documents; this difference however has no major impact on the results presented in this paper.

To recapitulate the language, consider this event query program consisting of two deductive rules ("views"):

```
F(x) <- c: C(x), d: D(x), e: E(x),
        c before d, {c,d} within 4 hours,
        d before e, {d,e} within 1 hour
C(x) <- a: A(x), b: B(x), {a,b} within 2 hours
```

The second rule generates an event $C(x)$ whenever events $A(x)$ and $B(x)$ happen within 2 hours (with the same value for x). Note that events $A(x)$ and $B(x)$ can happen over time intervals (not just points), and the occurrence time of $C(x)$ is the interval covering both of its constituent events. The first rule generates an event $F(x)$ from events $C(x)$, $D(x)$, and $E(x)$, provided that the specified temporal conditions are met, i.e., $D(x)$ happens after $C(x)$ but no more than 4 hours, and $E(x)$ happens after $D(x)$ but no more than 1 hour. Note that the $C(x)$ used in the first rule is generated by the second rule.

Events in the rule body are prefixed with an identifier (e.g., `c`, `d`), which is used for referring to them in conditions expressing temporal relationships between events (e.g., `c before d`). As far as this work is concerned, temporal conditions are built with Allen's interval relations [3] (`before`, `during`, etc.) as well as a metric constraint limiting the overall duration of a set events, written as `{i,j,...} within x`.

In principle, the language also supports further temporal conditions, but these are either not helpful for temporal relevance (e.g., `{i,j} 2 hours apart`) or require more complicated temporal reasoning (e.g., constraints involving periodic time intervals such as `{i,j} within workday`, which are often also based on a domain-specific calendar). The latter issue is addressed in Section 6.1. We consider here only conjunctions in our rule bodies; disjunctions can be obtained by splitting a rule.[1] As we will see, our approach also covers advanced querying features such a negation (i.e., absence) and data aggregation (e.g., averages) over events

We consider in this work only hierarchical rule programs, i.e., programs that are free of cyclic recursion between rules. Note that this is a common restriction not just for event queries but also database views and queries, and causes no problems in most practical applications.

2.2 Complex Event Relational Algebra

Evaluating an event query program is a step-wise procedure: whenever a new event is received, all rules must be checked for new answers based on the new event and a (partial) history of previous, relevant events. In addition, each such step also involves maintenance work necessary for future evaluation steps such as inserting the new event into the history and materializing intermediate results. A core observation in [8] is that query evaluation can be based on relational algebra and that we can separate the algebraic query plan and its incremental evaluation.

We associate a relation R_i with each atomic event query $i : R(x_1, \ldots, x_n)$. Each event of type R that happens corresponds to one tuple in R_i. Its occurrence time interval is part of that tuple, and expressed with its starting time $i.s$ and ending time $i.e$ (where i is name of the event identifier bound in the atomic event query). Accordingly, R_i has the schema $sch(R_i) = \{i.s, i.e, x_1, \ldots, x_n\}$.

With this, rule bodies can be translated into a variant of relational algebra, called CERA (for complex event relational algebra) here. The core idea of CERA is to treat time stamps much like regular data, in particular, temporal conditions become simply selections and the usual semantics of relational algebra apply. CERA is expressive enough to cover the considered language.[2] It is restricted so that all possible CERA-expressions are "reasonable" for the step-wise evaluation over time of event queries. The latter point is our motivation for introducing CERA and will be made more

[1]The standard pattern from logic programming is to replace a rule $a \leftarrow b \vee c$ with two rules $a \leftarrow b$ and $a \leftarrow c$.

[2]See [8] as well as [7, 6] for a discussion that this expressivity is both desirable and sufficient for many practical applications.

precise and proven at the end of this section. Restrictions concern operations on time stamps.

In CERA, all schemas (of base relations, results of expressions and subexpressions) have at least one pair of time stamps (e.g., $i.s$ and $i.e$), and time stamps always occur pairwise (start and end). The subset of all time stamps in a schema $sch(E)$ of an expression is called the time schema and denoted $sch_{time}(E)$.

CERA allows the following operators: natural join ($\bowtie$), selection (σ), projection of data fields (π), and merging of time intervals (μ). For readability, we write parameters that are usually subscripts in square brackets, e.g., $\sigma[a < b]$ for $\sigma_{a<b}$. Projections are only allowed to discard data attributes (such as x); they are not allowed to discard time stamps (such as $i.s$). A merging operation $\mu[j \leftarrow i_1 \sqcup \cdots \sqcup i_n](E)$ computes a new occurrence time interval (with start and end time stamps $j.s$ and $j.e$) from existing occurrence times so that it covers all these intervals, i.e., $j.s = \min\{i_1.s, \ldots i_n.s\}$ and $j.e = \max\{i_1.e, \ldots i_n.e\}$. It further discards these time stamps (in the manner of a projection). Merging of time intervals is not really a new operation for relational algebra. It is equivalent to the following so-called extended projection [15], a common practical extension of relational algebra used to compute new attributes from existing ones:

$$\begin{aligned}\pi[\quad & j.s \leftarrow \min\{i_1.s, \ldots i_n.s\}, j.e \leftarrow \max\{i_1.e, \ldots i_n.e\},\\ & sch(E) \setminus \{i_1.s, \ldots i_n.s, i_1.e, \ldots i_n.e\} \quad](E)\end{aligned}$$

Example: Let C_c, D_d, E_e, respectively be the relations for the atomic event queries in the body of the first rule from earlier. It then corresponds to the relational algebra expression

$$\begin{aligned}&\mu[f \leftarrow c \sqcup d \sqcup e](\\ &\sigma[c.e < d.s](\\ &\sigma[\max\{c.e, d.e\} - \min\{c.s, d.s\} \leq 4](\\ &\sigma[d.e < e.s](\\ &\sigma[\max\{d.e, e.e\} - \min\{d.s, e.s\} \leq 1](\\ &\quad (C_c \bowtie D_d) \bowtie E_e \;)))))\end{aligned}$$

Note that all temporal conditions (such as `c before d` and `{c,d} within 4 hours`) from the query have been turned into selections. Because temporal information is simply data in tuples (as $c.s$, $c.e$, etc.), no special temporal operators or such are needed as part of the algebra; e.g., there is no need for a sequence operator as found in many event algebras.[3]

Additionally, CERA allows anti-semi-joins ($\overline{\ltimes}$) and grouping (γ) as long as they are restricted by some temporal window (as usual for negation and aggregation in complex event queries). For the problem of determining temporal relevance these operators add nothing new: an anti-semi-join can he treated just like a join (or θ-join, which in turn is a join and a selection) and grouping just like projection. We can therefore safely omit them here. We also omit the renaming of attribute names (β), since it is not relevant here. More on translating rule bodies into CERA can be found in [8].

We deviate in one point from [8]: There, relative timer events such as a timer event `w: extend(r,6h)` that is defined to extend the duration of an event `r: R` by 6 hours) are expressed using an extended projection: $\pi[w.s \leftarrow r.s, w.e \leftarrow r.e + 6](R)$. In CERA, this extended projection is not allowed. Instead, we use a join with an auxiliary relation X for the timer event: $R \bowtie X$, where $X := \{x \mid (x(r.s), x(r.e)) \in \pi[r.s, r.e](R), x(w.s) = x(r.s), x(w.e) = x(r.e) + 6\}$.[4] This X is a relation containing four time stamps: the timer event's $w.s$ and $w.e$, which are computed based on the time stamps $r.s$ and $r.e$ of R, and also $r.s$ and $r.e$, which are needed for joining. The solution with a join is preferable because it satisfies the temporal preservation property explained next.

Definition: The occurrence time of a tuple e in the result of an expression E is the latest time stamp in e, i.e., $m_E(e) := \max\{e(i.e) \mid i.e \in sch_{time}(E_k)\}$.

The shorthand $m_E(e)$ is introduced because we will need this longwinded expression fairly often. To refer to the occurrence time in selections we also introduce the shorthand $M_E := \max\{i_1.e, \ldots, i_n.e\}$ where $\{i_1.e, \ldots, i_n.e\} \cup \{i_1.s, \ldots, i_n.s\} = sch_{time}(E)$. M_E is basically the same as m_e, only on the syntactic level of the algebra (as needed in conditions of selections) instead of the semantic level of individual tuples.

Note that in translating event queries into CERA, we pretend to have a kind of "omniscience" that we will not have in the actual query evaluation: the relations contain conceptually *all* events that ever happen. In the actual evaluation of event queries, on the other hand, we have to work with histories of streams, where we have no way of knowing future events. The restrictions CERA imposes on expressions (as compared to an unrestricted relational algebra), make this approach reasonable since we do not need any knowledge about future events when we want to obtain all results of an expression with an occurrence time until now. More precisely, to compute all results of E with an occurrence time before or at time point *now*, we need to know its input relations only up to this time point *now* as following theorem shows.

Theorem of temporal preservation: Let E be an CERA-expression with input relations $R_1, \ldots, R_n$. Then for all time points *now*: $\sigma[M_E \leq now](E) = E'$, where E' is obtained from E be replacing each R_k with $R'_k := \{r \in R_k \mid m_E(r) \leq now\}$.

Proof sketch: By induction. For Selection, projection, and join, the claim is obvious since time stamps are not changed at all. Merging does, by definition, not change the maximum value over all time stamps. Anti-semi-join and grouping are only allowed with a temporal restriction (cf. [8]) that also ensures that the maximum value is maintained.

2.3 Query Plans with Materialization Points

The translation of queries in rule bodies into CERA does not account for (1) the flow of information from one rule to another, i.e., chaining of rules, and (2) which intermediate results will be materialized to avoid their re-computation in later evaluation steps. We address this by introducing query plans with materialization points.

Definition: A query plan is a sequence $QP = \langle Q_1 := E_1, \ldots, Q_n := E_n \rangle$ of materialization point definitions $Q_i := E_i$, where Q_i is a relation name and E_i either a CERA-expression or a union $R_1 \cup \cdots \cup R_n$ of relations (both base relations and/or materialization points).[5] Each relation name Q_i is defined only once in QP, i.e., $Q_i \neq Q_j$ for all $i \neq j$.

[3] Join and selection can be combined into a θ-join, which has no special consequences for determining temporal relevance but gives the efficient evaluation of a sequence operator.

[4] $x(y)$ denotes the value for attribute y of a tuple x as usual.

[5] Unions are needed for translating rule programs such as `A <- b:B, c:C; A <- b:B, d:D`. For determining tempo-

Query Plan 1:

$$\begin{array}{lll} C_c & := & \mu[c \leftarrow a \sqcup b](\\ & & \sigma[\max\{a.e, b.e\} - \min\{a.s, b.s\} \leq 2](\\ & & \quad A_a \bowtie B_b \;)) \\ F_f & := & \mu[f \leftarrow c \sqcup d \sqcup e](\\ & & \sigma[c.e < d.s](\\ & & \sigma[\max\{c.e, d.e\} - \min\{c.s, d.s\} \leq 4](\\ & & \sigma[d.e < e.s](\\ & & \sigma[\max\{d.e, e.e\} - \min\{d.s, e.s\} \leq 1](\\ & & \quad (C_c \bowtie D_d) \bowtie E_e \;))))) \end{array}$$

Query Plan 2:

$$\begin{array}{lll} C_c & := & \mu[c \leftarrow a \sqcup b](\\ & & \sigma[\max\{a.e, b.e\} - \min\{a.s, b.s\} \leq 2](\\ & & \quad A_a \bowtie B_b \;)) \\ V_{c,d} & := & \sigma[c.e < d.s](\\ & & \sigma[\max\{c.e, d.e\} - \min\{c.s, d.s\} \leq 4](\\ & & \quad C_c \bowtie D_d \;)) \\ F_f & := & \mu[f \leftarrow c \sqcup d \sqcup e](\\ & & \sigma[d.e < e.s](\\ & & \sigma[\max\{d.e, e.e\} - \min\{d.s, e.s\} \leq 1](\\ & & \quad V_{c,d} \bowtie E_e \;))) \end{array}$$

Figure 1: Example Query Plans

The definitions must be acyclic, i.e., for all i, j, if Q_j occurs in E_i then $j < i$.

Since a query plan is acyclic, its semantics are straightforward: compute the results of its expressions from left right, replacing references to materialization points with their (already computed) result. The temporal preservation of CERA continues to hold for query plans.

Example: We illustrate query plans using our example rule program. Relation C_c in our CERA expression from earlier is not a base relation but computed by another rule. By introducing a materialization point C_c as in query plan 1 of Figure 1 and computing sequentially, i.e., first C_c and then F_f using C_c, we address the flow of information between rules, issue (1) from above. For issue (2), consider that the join $C_c \bowtie D_d$ is potentially re-computed in the incremental evaluation whenever a new event in E_e arrives (cf. next Section). To indicate that we materialize it as an intermediate result, we split the expression of F_f by introducing another materialization point $V_{c,d}$ as in query plan 2 of Figure 1.

So far it might seem that this is just an insignificant change in notation. However, it will become clear in the next section that only those intermediate results are "materialized," i.e., remembered across individual evaluation steps, that have a materialization point. Therefore the query plan 1 and 2 are different in terms of incremental evaluation, although of course both yield the same results for C_c and F_f. Note that the efficency of a query plan depends on characteristics of its event streams and there is no general principle to tell which one is more efficient.

A further salient aspect of these query plans is that they account for multi-query optimization where shared subexpressions are evaluated only once. For example the common subexpression $T \bowtie U$ in $\langle Y := T \bowtie U \bowtie W, Z := T \bowtie U \bowtie Y\rangle$ can be shared by rewriting into the semantically equivalent plan $\langle V := T \bowtie U, Y := V \bowtie W, Z := V \bowtie Y\rangle$.

ral relevance it is convenient to have unions only at materialization points (not anywhere in CERA-expression) in order to distinguish, e.g., the two B relations.

$$\begin{array}{ll} \triangle\mu_M(E) & = \mu_M(\triangle E) \\ \triangle\pi_P(E) & = \pi_P(\triangle E) \\ \triangle\sigma_C(E) & = \sigma_C(\triangle E) \\ \triangle(E_1 \bowtie E_2) & = \triangle E_1 \bowtie \circ E_2 \cup \triangle E_1 \bowtie \triangle E_2 \cup \circ E_1 \bowtie \triangle E_2 \\ \triangle(E_1 \cup E_2) & = \triangle E_1 \cup \triangle E_2 \\ \circ\mu_M(E) & = \mu_M(\circ E) \\ \circ\pi_P(E) & = \pi_P(\circ E) \\ \circ\sigma_C(E) & = \sigma_C(\circ E) \\ \circ(E_1 \bowtie E_2) & = \circ E_1 \bowtie \circ E_2 \\ \circ(E_1 \cup E_2) & = \circ E_1 \cup \circ E_2 \end{array}$$

Figure 2: Equations for finite differencing

2.4 Incremental Evaluation

Evaluation of an event query program, or rather its query plan QP, over time is a step-wise procedure: A step is initiated by some base event (an event not defined by a rule) happening at the current time, which we denote *now*. For each materialization point Q in QP, the required output for this step then is all computed answers (tuples q representing materialized intermediate results and derived events) that "happen" at this current time *now*, i.e., where $m_Q(q) = now$. In other words in each step, we are not interested in the full result of Q, but only in $\triangle Q := \sigma[M_Q = now](Q)$. We assume for simplicity here that the base events are processed in the temporal order in which they happen, i.e., with ascending ending time stamps. Extensions where the order of events is "scrambled" (within a known bound) are possible, however. Note that while the time domain can be continuous (e.g., isomorphic to the real numbers), the number of evaluation steps is discrete since we assume a discrete number of incoming events.

A naive, non-incremental way of query evaluation would be: Maintain a stored version of each base event relation across steps. In each step simply insert the new event into its base relation and evaluate the query plan from scratch according to its non-incremental semantics (previous section). Then apply the selection $\sigma[M_Q = now]$ for each materialization point to output the result of the step. This is, however, inefficient since we compute not only the required result $\triangle Q = \sigma[M_Q = now](Q)$, but also all results from previous steps, i.e., also $\sigma[M_Q < now](Q)$.

It is more efficient to use an incremental approach, where we (1) store not only base relations but also some intermediate results, namely those of each materialization point Q across steps and then (2) in each step only compute the changes to Q that result from the step. It turns out that due to the temporal preservation of CERA, the change to each Q involves only inserting new tuples into Q and that these tuples are exactly the ones from $\triangle Q$.

We can compute $\triangle Q$ efficiently using the changes $\triangle R_i$ to the input relations R_i of $Q := E$, together with $\circ R_i = \sigma[M_Q < now](R_i)$, their materialized states from the previous evaluation step. Using finite differencing, we can derive a CERA-expression $\triangle E$ so that $\triangle E$ involves only $\triangle R_i$ and $\circ R_i$ and $\triangle E = \triangle Q$ (for each step). Finite differencing pushes the differencing operator $\triangle$ inwards according to the equations in Figure 2.

Finite differencing is a method originating in the incremental maintenance of materialized views in databases, which is a problem very similar to incremental event query evaluation. Note that the details of finite differencing are not of primary importance to determining temporal relevance, the subject of this paper. However it is necessary to have a general understanding of incremental evaluation. Especially it must be understood that, as explained, the results of all materialization points are stored across evaluation step (and no other intermediate results). We refer to [18, 8] for more information on finite differencing and incremental evaluation.

3. PROBLEM DEFINITION

If we were to evaluate query plans incrementally as described in the previous section without any notion of temporal relevance, we would have to store the full history of all event tuples for each base relation and materialization point. Our goal with temporal relevance is to limit the stored history of event tuples to only the necessary knowledge of the past (as needed for producing all answers).

Temporal relevance is in this sense symmetrical to temporal preservation: temporal preservation makes a statement about what knowledge of the future is needed for query evaluation, temporal relevance about knowledge of the past. However they are asymmetrical in that no knowledge whatsoever is needed about the future, while a significant amount of knowledge about the past can be needed. Further, temporal preservation is a property that is always given (by design of CERA), while temporal relevance is specific to a given query plan and has to be determined in some algorithmic way.

3.1 Relevance and Temporal Relevance

The need to store histories of event tuples comes from joins in CERA-expressions. Only joins combine event tuples received at different times. We can remove event tuples from the histories, when we know for sure that they will not affect any future answers, i.e., become irrelevant.

Definition: Let E be a CERA-expression with input relations $R_1, \dots R_n$. A tuple r of an input relation R_i is **relevant** for E at time *now*, if it might be joined (now or at a later time) with tuples from the other relations to produce an answer e to E with an occurrence time $m_E(e)$ of or later than *now*.

Note that the occurrence time of the answer implies that at least one of these other tuples has a time stamp $\geq now$. When a tuple joins with other tuples, it might actually not produce an answer for E. This is the case when the resulting tuple(s) of the join are eliminated by a selection in E.

Example: Consider the expression for C_c in Figure 1. Let $a = \{a.s = 9, a.e = 10, x = 42\}$ and $b = \{b.s = 11, b.e = 11, x = 42\}$ be tuples in A_a and B_b, respectively. At time $now = 12$, tuple a is not relevant anymore: any tuple r resulting from joining a with a B_b-tuple arriving at this or a later time (i.e., having $b.e \geq now = 12$) will be eliminated by the expression's selection $\sigma[\max\{a.e, b.e\} - \min\{a.s, b.s\} \leq 2]$ and thus not produce an answer. Tuple b is still (temporal) relevant at time $now = 12$, and also —to illustrate that the time domain is not limited to integers— at time $now = 12.3$. It becomes irrelevant when $now > 13$ due to the selection.

Temporal relevance is a particular form of relevance that is derived only from selections expressing temporal conditions and where we make no assumptions about what other tuples are currently stored or might arrive in the future. We revisit this in Section 6.3 and 6.4.

3.2 Temporal Relevance Conditions

Temporal relevance must be expressed in a formal way and in a restricted syntax so that we have (1) a clear notion of output of an algorithm that statically determines temporal relevance, (2) a basis for implementing garbage collection in query evaluation, and (3) means for correctness proofs. One might expect that temporal relevance could be expressed just as time window for each relation that states how long each tuple must be kept. However, since tuples have several time stamps —which is necessary for expressive event queries— this is not sufficient, and we express temporal relevance as so-called temporal relevance conditions. Conditions have also the advantage that they generalize well to other forms of relevance (cf. Sections 6.1 and 6.4).

For each materialization point Q in a given query plan QP, and each input relation R (base relation or materialization point) of Q, we will have a temporal relevance condition $TR_{R \text{ in } Q}$. If $TR_{R \text{ in } Q}$ is true for a tuple $r \in R$ at the current time *now*, then this tuple is still relevant. If it is false, the tuple is irrelevant and can be removed by the garbage collector. Removing is "optional," i.e., irrelevant tuples that are not removed do not disturb query evaluation. Temporal relevance conditions derived in this work have the following form:

$$\begin{aligned} TR_{R \text{ in } Q} \equiv \quad & i_1 \geq now - rt_1 \wedge \dots \wedge i_m \geq now - rt_m \wedge \\ & i_{m+1} > now - rt_{m+1} \wedge \dots \wedge i_n > now - rt_n \end{aligned}$$

where $\{i_1, \dots i_n\} \subseteq sch_{time}(R)$ are some time stamps of R. All rt_k are fixed durations (lengths of time) and can be understood as individual time windows for each time stamp in the condition. A rt_k is called the relevance time of time stamp i_k, and also written $rt(i_k)$ or, making explicit the input relation R and the expression or materialization point Q, $rt_{R \text{ in } Q}(i_k)$. Note that the syntax of temporal relevance conditions is such that they can be used as a conditions of selection operators when we read *now* as a constant expressing the current time. This is convenient because garbage collection can be implemented using existing functionalities from selection operators.

Definition: Let QP be a query plan and $K = \{TR_{R \text{ in } Q}, \dots\}$ a set of temporal relevance conditions, one for each input relation R of each materialization point $Q := E$. K is correct if for all possible base relations of QP the following holds for each time point *now* and each materialization point $Q := E$ of QP: all current (at time *now*) and future results of E are the same as those of E', i.e., $\sigma[M_E \geq now](E) = \sigma[M_{E'} \geq now](E')$, where E' is obtained from E by replacing each input relation R (base relation or materialization point) with any R' such that $\{r \in R \mid TR_{R \text{ in } Q}(r)\} \subseteq R' \subseteq R$.

Note that when the input relation R is a materialization point, then R' derives from the definition of $R := E_R$ in the *original* query plan QP, and *not* from some "E'_R" where in turn a replacing of the input relations of E_R has taken place.

The intuition of correctness is simple: the temporal relevance conditions are correct when removing tuples that are, according to the conditions, deemed irrelevant from the event histories (which corresponds to replacing the full history R with R') does not influence the query result at the current time *now* or in the future. We do not care about past results because they have already been produced in earlier evaluation steps. Note that since $\sigma[M_E \geq now](E) = \sigma[M_{E'} \geq now](E')$ must hold for all possible times *now*, we

Query Plan 1:

$$\begin{aligned}
TR_{A_a \text{ in } C_c} &\equiv a.s \geq now - 2 \wedge a.e \geq now - 2\\
TR_{B_b \text{ in } C_c} &\equiv b.s \geq now - 2 \wedge b.e \geq now - 2\\
TR_{C_C \text{ in } F_f} &\equiv c.s \geq now - 5 \wedge c.e \geq now - 5\\
TR_{D_d \text{ in } F_f} &\equiv d.s \geq now - 1 \wedge d.e \geq now - 1\\
TR_{E_e \text{ in } F_f} &\equiv e.s > now - 1 \wedge e.e \geq now - 0
\end{aligned}$$

Query Plan 2:

$$\begin{aligned}
TR_{A_a \text{ in } C_c} &\equiv a.s \geq now - 2\\
TR_{B_b \text{ in } C_c} &\equiv b.s \geq now - 2\\
TR_{C_c \text{ in } V_{c,d}} &\equiv c.s \geq now - 4\\
TR_{D_d \text{ in } V_{c,d}} &\equiv d.s > now - 4 \wedge d.e \geq now - 0\\
TR_{V_{c,d} \text{ in } F_f} &\equiv d.s \geq now - 1\\
TR_{E_e \text{ in } F_f} &\equiv e.s > now - 1 \wedge e.e \geq now - 0
\end{aligned}$$

Figure 3: Temporal Relevance Conditions

could equally write $\sigma[M_E = now](E) = \sigma[M_{E'} = now](E')$ in the definition.

Example: The temporal relevance conditions for the query plans from Figure 1 are shown in Figure 3. The first four conditions of query plan 1 are longer than they need to be: the comparison $a.e \geq now - 2$ is superfluous since it is implied by $a.s \geq now - 2$ and the temporal conditions of the query expression (analogous for $b.e$ and $c.e$). For query plan 2, all conditions are minimal in the sense that they contain no superfluous comparisons. For both query plans, each condition is optimal in the sense that the time windows cannot be made tighter for any time stamp.

4. SOLUTION

Our task is to determine the relevance time $rt_{R \text{ in } E}(i)$ of each time stamp i of an input relation in a given expression E. We focus in this section on the general idea and give a full algorithm in the next section. Note that in the following when we speak of a "time stamp" we always mean the element of the schema (e.g., i), not a concrete value of a given tuple (e.g., $r(i)$). This is natural since our method runs at compile time where we do not have any concrete values but reason abstractly about all possible values.

4.1 Temporal Distances

For determining relevance times, we use an auxiliary computation. For each pair i, j of time stamps in the expression E, we establish from the temporal selections in E an upper bound $td(i,j) \geq 0$ such that any tuples in the result of E will obey $j - i \leq td(i,j)$. The upper bound depends on the temporal selections in E and we want it to be as small as possible. For reasons that will become clear shortly, we call this least upper bound the temporal distance from i to j. The temporal distances from i to j and from j to i can be different. For example in the expression of F_f of query plan 1, $td(c.e, d.s) = 4$ but, since c must happen before d, $td(d.s, c.e) = 0$.

For a given time stamp i, its relevance time $rt(i)$ then simply is the longest of all temporal distances $td(i,j)$ from i to some other time stamp j of E, i.e.,

$$rt(i) = \max\{td(i,j) \mid j \in sch_{time}(E)\}$$

This equation for the relevance time can be explained as follows. Let the current time be now. Consider a time stamp i for a tuple r_1 that is stored in an input relation R_1 of expression E. Note that since the tuple is already stored, the value for the time stamp i is $r_1(i) < now$. Now let another tuple r_2 of another relation R_2 arrive at the current time now, i.e., $m_E(r_2) = now$. Let the two tuples join (together with other tuples from other relations in case there are more than two input relations) to yield a tuple e. Let $j \in sch_{time}(R_2)$ be a time stamp such that its value $r_2(j) = now$; since $m_E(r_2) = now$ such a j must exist. Suppose now that $r_1(i) < now - rt(i)$, i.e., r_1 is deemed irrelevant according to its relevance time. Then $r_2(j) - r_1(i) > rt(i)$, and thus also $e(j) - e(i) > rt(i)$. Construction of $rt(i)$ ensures that there must be a selection in E that eliminates e. Note that this selection does *not* necessarily involve either i or j.

4.2 Temporal Distance Graph

It turns out that all temporal conditions in selections (cf. Section 2.1 and 2.2) are equivalent to a conjuction of comparisons of the form $j - i < t$ or $j - i \leq t$, where i, j are time stamps. For example, $\max\{a.e, b.e\} - \min\{a.s, b.s\} \leq 2$ (translated from `{a,b} within 2 hours`) is equivalent to $b.e - a.s \leq 2 \wedge a.e - b.s \leq 2 \wedge a.e - a.s \leq 2 \wedge b.e - b.s \leq 2$. Similarly, $c.e < d.s$ (from `c before d`) gives us $d.s - c.e < 0$. To simplify explanation, we assume for now that we only have conditions of the form $j - i \leq t$. The extension to $<$ later is simple and given in Section 4.3.

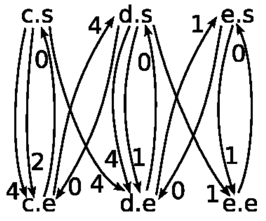
This observation is very helpful because it allows us to frame the problem of computing all temporal distances in an expression E as an "all pairs shortest paths" (APSP) problem in a directed, weighted graph. We refer to the graph we construct as the *temporal distance graph* (TDG) of an expression E. Its nodes are all the time stamps occurring in E. Each temporal condition $j - i \leq t$ in a selection of E generates a directed edge from node i to j with weight t.

Further, for each pair $i.s, i.e$ of time stamps, we generate an edge with weight 0 from $i.e$ to $i.s$. These edges reflect that the ending time of an event cannot be before its starting time ($i.s \leq i.e$). We will later see that edges will also be generated for information known about input relations or for merging of time intervals.

Since the temporal distance $td(i,j)$ between two time stamps is the least upper bound t so that $j - i \leq t$, it corresponds to the (length of the) shortest path from i to j in the graph. The reason for this is that, like path length in a graph, bounds between time stamps obey the triangle equality, i.e., from $k - j \leq t_1$ and $j - i \leq t_2$ we can conclude that $k - i \leq t_1 + t_2$.

The matrix of all shortest paths between all time stamps in E can be computed using a standard algorithm such as Floyd-Warshall [21]. The relevance time $rt(i)$ for i then is simply the maximum entry in the row corresponding to i in this matrix.

Example: Figure 4 shows the temporal distance graph for the expression of F_f of query plan 1 and the corresponding matrix of shortest paths. Compare the maximum entries in each matrix row with the relevance times of Figure 3. The graph does not (yet) include nodes for the time stamps $f.s$ and $f.e$. It has an edge from $c.s$ to $c.e$ with weight 2, which reflects knowledge about input relation C_c. Details for this will be provided shortly. Note that it is not the general case that the lower-left half of the matrix is 0; this is only because this particular query demands the events c, d, e occur sequentially.



from\to	$c.s$	$c.e$	$d.s$	$d.e$	$e.s$	$e.e$
$c.s$	0	2	4	4	5	5
$c.e$	0	0	4	4	5	5
$d.s$	0	0	0	1	1	1
$d.e$	0	0	0	0	1	1
$e.s$	0	0	0	0	0	1
$e.e$	0	0	0	0	0	0

Figure 4: Temporal distance graph, shortest paths

$$\begin{array}{ll}
(<,x)+(<,y)=(<,x+y) & (<,x)<(<,y) \iff x<y \\
(<,x)+(\le,y)=(<,x+y) & (<,x)<(\le,y) \iff x\le y \\
(\le,x)+(<,y)=(<,x+y) & (\le,x)<(<,y) \iff x<y \\
(\le,x)+(\le,y)=(\le,x+y) & (\le,x)<(\le,y) \iff x<y
\end{array}$$

Figure 5: Strict and (non-)strict edges

Having presented the general idea, the remainder of this section refines it with some necessary details, before the full algorithm is given in the next section.

4.3 Strict Inequalities

So far we have pretended that we have only non-strict inequalities such as $j - i \le t$ and ignored the fact that some are in fact strict, i.e., $j - i \le t$. We suggest two ways to deal with strict inequalities.

First, we can simply treat all strict inequalities as non-strict when building the temporal distance graph, as we have done so far. (Of course, in the evaluation of the query plan, we will distinguish $<$ and $\le$.) This is leads to a marginal "over-estimation" on the temporal relevance conditions and we will store in some cases more tuples than necessary. Since we only store *more* tuples, the result of the real evaluation is still correct. Because we store only very little more, the consequences on performance are usually minimal.

Second, we can explicitly distinguish $<$ and $\le$ by "extending" our time domain $\mathbb{T}$ to $\{\le, <\} \times \mathbb{T}$. The duration $(\le, t)$ simply corresponds to the "old" t. Its sister duration $(<, t)$ can be imagined as $t - \epsilon$ with an infinitesimally small ϵ. Conditions such as $i - j < t$ can then be read as $i - j \le (<, t)$. The operation $+$ and the relation $<$ on durations, which are needed in the computation of shortest paths and relevance times, must be adapted appropriately, as shown in Figure 5. Note that the relation $<$ automatically gives a definition for min and max operations on the extended time domain.

When we have determined the relevance time $rt(i)$ for a time stamp i (as maximum of i's row in the matrix), the first component ($<$ or $\le$) in the tuple of will indicate whether the corresponding temporal relevance condition contains $i > now - w$ (in case $rt(i) = (<, w)$) or $i \ge now - w$ (in case $rt(i) = (\le, w)$).

4.4 Recognizing Superfluous Relevance Times

In Figure 3, the first four temporal relevance conditions of query plan 1 contain superfluous comparisons: each right comparison is implied by the left. It is desirable for performance to avoid such unnecessary comparisons and have minimal temporal relevance conditions as those of query plan 2.

We can remove implied comparisons in a post processing. To do this, we must distinguish whether an edge in a temporal distance graph is already guaranteed to hold for all time stamps by the input relation (i.e., independently of the expression) or whether it has been added by some selection of the expression (and must be tested in the query plan evaluation). We will mark guaranteed edges with an exclamation mark (!) and non-guaranteed edges with a question mark (?). Now we can define when a relevance time of a time stamp covers the relevance time of another time stamp to express that the comparison of the covered one is unnecessary.

$$\begin{array}{ll}
(?,x)+(?,y)=(?,x+y) & (?,x)<(?,y) \iff x<y \\
(?,x)+(!,y)=(?,x+y) & (?,x)<(!,y) \iff x<y \\
(!,x)+(?,y)=(?,x+y) & (!,x)<(?,y) \iff x\le y \\
(!,x)+(!,y)=(!,x+y) & (!,x)<(!,y) \iff x<y
\end{array}$$

Figure 6: Guaranteed and non-guaranteed edges

Definition: The relevance time $rt(i)$ of a time stamp i is said to *cover* the relevance time $rt(j)$ of another time stamp j, if the shortest path from j to i in the temporal distance graph uses only edges marked as guaranteed and for its length $td(j, i)$ it holds that $td(j, i) + rt(i) = rt(j)$.

This can be explained as follows: Since the shortest path from j to i uses only guaranteed edges, $i - j \le td(j, i)$ is true for any tuple in the input relation. Then, $i - j + rt(i) \le rt(i) + td(j, i) = rt(j)$ (assumption) and $i \ge now - rt(i)$ together imply that $j \ge i + rt(i) - rt(j) \ge now - rt(j)$.

Commonly when solving the all pairs shortest path (APSP) problem we compute only the *length* of the shortest paths and not the paths themselves. The definition above however refers to the shortest paths itself to test whether they use only guaranteed edges. It turns out that we do not need the actual shortest paths but can make the information about guaranteed and non-guaranteed edges part of the time domain, much like we did with the $<$ and $\le$ markings earlier. The corresponding definitions for addition and comparison are given in Figure 6. Note that the time domain that is extended can be already $\{\le, <\} \times \mathbb{T}$ from the previous section.

4.5 Merging of Time Intervals

In the example of Figure 4, we have neglected time stamps $f.s$ and $f.e$, which are generated by the merging operation (μ) in F_f of query plan 1. One might be tempted to disregard such time stamps in the temporal distance graph since they do not occur in any of the temporal relevance conditions. However, (1) a temporal selection such as $\sigma[f.e - f.s < 1]$ could be performed after the merging and its effect should be propagated to the time intervals f has merged (in the example: c, d, e); and (2) knowledge about a time stamp generated by merging (e.g., c in the expression for C_c) might be needed in other expressions (e.g., $c.e - c.s < 2$ in F_f).

To include time stamps $j.s$ and $j.e$ generated by a merging $\mu[j \leftarrow i_1, \ldots i_n](E)$ into the temporal distance graph, we add them as nodes and add edges as follows: Since $j.s = \min\{i_1.s, \ldots i_n.s\}$, it holds that $j.s - i_k.s \le 0$ for $k = 1..n$ and we add corresponding edges into the graph, which are marked as guaranteed (!). Analogously for $j.e = \max\{i_1.e, \ldots i_n.e\}$, we add guaranteed edges for $i_k.e - j.e \le 0$ $(k = 1..n)$. Further we add an edge from $j.s$ to $j.e$, its length being the longest of all temporal distances between any $i_k.s$ and $i_l.e$.

The edges between $j.s$ and the $i_k.s$, and between $j.e$ and the $i_k.e$ will propagate any temporal selection involving $j.s$ or $j.e$ "down" to the i_k. The length of the edge from $j.s$ to $j.e$ is justified by the definition of $j.s$ and $j.e$ as minimum and maximum. As will become clear in the full algorithm (cf. Section 5), the length can be computed immediately when $j.s$ and $j.e$ are added to the temporal distance graph since the merging operator discards the $i_k.s$ and $i_k.e$, and accordingly the edges between them are all already known.

4.6 Propagation in Query Plans

The temporal distance graph for a given expression E should make use of knowledge about input relations. An example is the constraint that $c.e - c.s < 2$ in the graph of Figure 4, which derives from the expression that computes the input relation C_c as a materialization point. When E has an input relation that is defined by a materialization point Q, the temporal distance graph for E should contain the subgraph of Q's temporal distance graph that contains only the nodes of $sch_{time}(Q)$ and edges between them. These edges should all marked as guaranteed (!) now in E's temporal distance graph since they are automatically satisfied.

Since query plans are acyclic, we can compute temporal relevance conditions from left to right so that Q's temporal distance graph is readily available when we have to compute the one for E.

4.7 Unions in Materialization Points

Recall that CERA does not have a unions, but that they can be expressed in materialization points of the form $Q := R_1 \cup \cdots \cup R_n$, where each R_i is either a base relation or a materialization point. We do not need to derive temporal relevance conditions for the input relations R_i in unions since there is no need to store them (cf. Section 2.4). However expressions referring to Q as an input relation need a temporal distance graph for Q (cf. previous section).

The temporal distance graph for Q derives from the temporal distance graphs of the input relations $R_1, \ldots R_n$. The nodes are the time stamps of $sch_{time}(Q)$; note that $sch_{time}(Q) = sch_{time}(R_1) = \cdots = sch_{time}(R_n)$. Each edge from i to j in the temporal distance graph of Q is assigned the maximum of the individual lengths of the shortest paths between from i to j in the temporal distance graphs of the input relations $R_1, \ldots R_n$. Keep in mind that the length of this shortest path can be ∞, and an edge with weight ∞ is the same as having no edge.

Having unions only in this limited form at materialization points, not in CERA itself makes our explanations and the algorithm in the next section somewhat simpler. Consider an expression such as $Q := \sigma[\max\{i.e, j.e\} - \min\{i.s, j.s\} < 2](R_1 \cup R_2)$, which is not allowed in CERA. The temporal distance graph for it would contain only four time stamps $i.s$, $i.e$, $j.s$, $j.e$. However we should distinguish the time stamps of R_1 and R_2, since they can give different temporal relevance conditions. For example, R_1 might guarantee (from its definition as a materialization point) that $i.e < j.s$, while R_2 does not guarantee this. Then $TR_{R_1 \text{ in } Q} \equiv i.s \geq now - 2$ is correct, but for R_2 the relevance time of $j.s$ is not covered by that of $i.s$, i.e., $TR_{R_2 \text{ in } Q} \equiv i.s \geq now - 2 \wedge j.s \geq now$.

An alternative to restricting CERA as done here would be to distinguish time stamps of input relations, i.e., have eight time stamps $i_{R_1}.s$, $i_{R_2}.s$, … in the example above. Once the basic approach of this article is understood, making the adaption to this alternative is not hard. However, for the purpose of this article we found that it makes explanations and notation unnecessarily longwinded.

5. ALGORITHM

We now give an algorithm computing all temporal relevance conditions for a query plan QP. The algorithm is specified on a high abstraction level with mathematical functions. Its implementation in a functional programming language (Haskell) mirrors the specification very closely. An implementation in an imperative language is also straightforward. Reader comfortable with the explanations of the previous section and less inclined to the algorithmic details can skip ahead to Section 5.4.

5.1 Computing the Temporal Distance Graph

The temporal distance graph $tdg_{QP}(E)$ of an expression E in the context of a query plan QP is a directed graph $G = (V_G, E_G)$ with weighted edges. Its vertices are all time stamps occurring in E. Its edges have temporal distances (marked as strict or non-strict, and guaranteed or non-guaranteed) as weights, i.e., $E_G \subseteq V_G \times \mathbb{W} \times V_G$ with $\mathbb{W} = \{?, !\} \times \{<, \leq\} \times \mathbb{T}$. Note that we allow multiple edges with different weights between the same two nodes; however since we are eventually only interested in shortest paths, only the edge with the least weight is relevant.

We use two auxiliary functions. The length of the shortest path between nodes i and j in a temporal distance graph G is given with $sp(G, i, j) \in \mathbb{W}$. The addition operation and order relation on path lengths, which are necessary to define and compute shortest path lengths, have been given in Figures 5 and 6. The function g operates on edge weights, turning a non-guaranteed edge (?) into a guaranteed edge (!), i.e., $g(?, <, t) = (!, <, t)$, $g(!, <, t) = (!, <, t)$, etc.

The computation of the temporal distance graph $tdg_{QP}(E)$ can be understood as an abstract interpretation (or pseudo-evaluation) of the expression E: the expression is evaluated in the same manner as usual, but on a different domain and with different interpretations of the operators. Instead of the standard domain (sets of tuples) we use a so-called abstract domain (temporal distance graphs) and instead of the standard interpretation of each operator (e.g., a join on relations) we use an abstract interpretation (e.g., for a join the union of graphs). Importantly, elements of the abstract domain correspond to ("are an abstraction of") sets of elements from the standard domain: a temporal distance graph corresponds to all relations whose tuples obey the restrictions set forth in the graph (e.g., if $(i, !, <, t, j)$ is an edge of $tdg(E)$ then in all possible results of E, all tuples obey $j - i < t$).

Keep in mind however that most of the literature on abstract interpretation focuses on the analysis of imperative programs, whereas we analyze relational queries. In comparison to abstract interpretation of imperative programs, computing temporal distance graphs is much simpler.

Like the standard interpretation of expressions, the temporal distance graph is defined inductively on the structure of expressions (with an additional case for unions of materialization points):

- $tdg_{QP}(R) = (V_G, E_G)$ for a base relation R with $sch_{time}(R) = \{i.s, i.e\}$, where
  $V_G = \{i.s, i.e\}$
  $E_G = \{(i.e, !, \leq, 0, i.s)\}$

- $tdg_{QP}(Q) = (V_G, E_G)$ for a materialization point Q defined as $Q := E$ in QP, where
  $(V'_G, E'_G) = tdg_{QP}(E)$

$V_G = sch_{time}(E)$
$E_G = \{(i, w, j) \mid i \in V_G, j \in V_G, (i, w, j) \in E'_G\}$

- $tdg_{QP}(R_1 \cup \cdots \cup R_n) = (V_G, E_G)$, where
  $V_G = sch_{time}(R_1) = \cdots = sch_{time}(R_n)$
  $E_G = \{(i, \max_{k=1..n} sp(tdg_{QP}(R_k), i, j), j) \mid i, j \in V_G\}$
- $tdg_{QP}(E_1 \bowtie E_2) = (V_G^1 \cup V_G^2, E_G^1 \cup E_G^2)$, where
  $(V_G^1, E_G^1) = tdg_{QP}(E_1)$
  $(V_G^2, E_G^2) = tdg_{QP}(E_2)$
- $tdg_{QP}(\mu[j \leftarrow i_1 \sqcup \cdots \sqcup i_n](E)) = (V_G, E_G)$, where
  $(V'_G, E'_G) = tdg_{QP}(E)$
  $w = \max\{sp(tdg_{QP}(E), i_k.s, i_l.e) \mid k = 1..n, l = 1..n\}$
  $V_G = V'_G \cup \{j.s, j.e\}$
  $E_G = E'_G \cup \{(j.s, !, w, j.e), (j.e, !, \leq, 0, j.s)\}$
  $\cup \{(i_k.s, !, \leq, 0, j.s) \mid k = 1..n\}$
  $\cup \{(j.e, !, \leq, 0, i_k.e) \mid k = 1..n\}$
- $tdg_{QP}(\pi[X](E)) = tdg_{QP}(E)$
- $tdg_{QP}(\sigma[C](E)) = tdg_{QP}(E)$ for a non-temporal condition C (i.e., a condition not involving time stamps)
- $tdg_{QP}(\sigma[i - j \leq t](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j, ?, \leq, t, i)\}$
- $tdg_{QP}(\sigma[i - j < t](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j, ?, <, t, i)\}$
- $tdg_{QP}(\sigma[\max\{i_1, \ldots, i_m\} - \min\{j_1, \ldots, j_n\} \leq t](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j_k, ?, \leq, t, i_l) \mid k = 1..n, l = 1..n\}$
- $tdg_{QP}(\sigma[i \leq j](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j, ?, \leq, 0, i)\}$
- $tdg_{QP}(\sigma[i < j](E)) = (V'_G, E_G)$, where
  $(V'_G, E'_G) = tdg_P(E)$
  $E_G = E'_G \cup \{(j, ?, <, 0, i)\}$

Note that for selection, the last three cases are just variations of the two cases before them and are given for the sake of completeness.

5.2 Computing Relevance Times and Temporal Relevance Conditions

Function $rt_{QP}(i, R, E)$ computes the temporal relevance time $rt_{R \text{ in } E}(i)$ of a time stamp $i \in sch_{time}(R)$ of an input relation R of an expression E or materialization point $Q := E$ in a given query plan QP:

- $rt_{QP}(i, R, E) = \max\{sp(tdg_{QP}(E), i, j) \mid j \in J\}$, where
  $R_1, \ldots, R_n$ are the input relations of E
  $J = sch_{time}(R_1) \cup \cdots \cup sch_{time}(R_n)$

Function $trs(QP)$ spells out all temporal relevance conditions for a given query plan QP in a straight-forward manner based on the relevance times. These temporal relevance conditions are still "maximal," i.e., contain also superfluous comparisons. It uses auxiliary functions $tr_{QP}(Q)$, which gives the temporal relevance condition of a single materialization point $Q := E$, and $comp(rt, i)$, which spells out the comparison in a temporal relevance condition corresponding to a given relevance time rt of a time stamp i.

- $trs(QP) = \bigcup_{Q:=E \in QP} tr_{QP}(Q)$
- $tr_{QP}(Q := E) =$
  $\{TR_{R \text{ in } Q} \equiv \bigwedge_{i \in sch_{time}(R)} comp(rt_{QP}(i, R, E), i) \mid$
  R is an input relation of $E\}$
- $comp(?, \leq, t, i) = i \geq now - t$
  $comp(?, <, t, i) = i > now - t$
  $comp(!, \leq, t, i) = i \geq now - t$
  $comp(!, <, t, i) = i > now - t$

5.3 Minimal Temporal Relevance Conditions

To obtain minimal temporal relevance conditions, which do not contain superfluous, "covered," comparisons, we define the boolean-valued function $covers_{QP}(R, E, i, j)$ in analogy of the definition in Section 4.4.

- $covers_{QP}(R, E, i, j) \iff sp(tdg_Q P(E), j, i) = (!, \cdot) \wedge$
  $sp(tdg_Q P(E), j, i) + rt_{QP}(i, R, E) = rt_{QP}(j, R, E)$.

Note that there is, in some cases, not a single unique minimal storage condition. This occurs for example when two time stamps of an input relation are guaranteed to happen at the same time, i.e., $i = j$, and thus each covers the other. Unless both i and j are covered by some third time stamp, we can remove either but not both from the temporal relevance condition.

In practice this doesn't cause any problems, because it does not matter which one we remove and there usually is a natural tie breaker like the processing order of time stamps (e.g., in the for each loop below). Function $trmin_{QP}(Q := E)$ is the analog of $tr_{QP}(Q := E)$ with the difference that it gives a *minimal* storage condition. The set $S \subseteq sch_{time}(R)$ of time stamps where no time stamp in S covers another in S is used in its definition and can, e.g., be obtained with the given procedure.

- $trmin_{QP}(Q := E) =$
  $\{TR_{R \text{ in } Q} \equiv \bigwedge_{i \in S} comp(rt_{QP}(i, R, E), i) \mid$
  R is an input relation of $E\}$
- Computation of S:

```
$S \leftarrow \emptyset$
For each $i \in sch_{time}(R)$:
    If $\neg\exists j \in S\ covers_{QP}(R, E, j, i)$
        $S \leftarrow (S \setminus \{j \in S \mid covers_{QP}(R, E, i, j)\} \cup \{i\}$
```

5.4 Complexity

The proposed algorithm can be implemented as a linear pass over the query plan with one step for each materialization point in it. Each step involves solving one all pairs shortest path problem. Using the Floyd-Warshall algorithm this can be done in time cubic in the number of time stamps occurring in the materialization point. Note that the intermediate shortest paths computation needed at a merging operator is part of solving the full all pairs shortest paths problem. Hence, it can be implemented in such a way that the work done there is saved in the later shortest paths computations and does not affect complexity.

While there are also subcubic algorithms for computing shortest paths, their associated overhead entails that they usually will not pay off in a practical implementation since the number of time stamps is too small (say < 20).

Our algorithm's complexity is easily acceptable because it is run only once during query compilation and because query compilation contains far more expensive operations. It might be interesting however, e.g., for branch and bound optimization of query plans, to investigate dynamic algorithms that can efficiently compute the effect of changes to an existing query plan on temporal relevance conditions or temporal distance graphs.

5.5 Correctness

Due to space restrictions we only sketch the correctness proof here; the core ideas have been given already in Section 4. By induction on the structure of expressions, one shows that the temporal distance constraints laid out by the computed temporal distance graph are satisfied by all possible results of an expression. That is, if the temporal distance graph contains an edge for $i - j < t$ or $i - j \leq t$ then all possible tuples in a result satisfy this. The argument of the end of Section 4.1 explains the correctness of the temporal relevance conditions derived from the temporal distance graph.

6. OUTLOOK

We have provided a concrete solution for statically determining temporal relevance. Since relevance in complex event queries is a novel issue raised in this article, we now broaden the scope and discuss variations and generalizations.

6.1 Generalization of Temporal Relevance

We have made the assumption that all temporal conditions can be written as a conjunction of comparisons of the form $i - j < t$ or $i - j \leq t$. This covers a significant, and for many practical applications sufficient, spectrum of possible temporal conditions. Not covered are, for example, conditions involving periodic time intervals such as "workday" (defined, e.g., as Monday through Friday, 9am to 5pm, except holidays) which are often also based on domain-specific calendars (cf. Section 2.1).

Our algorithm can still be used by "approximating" these conditions. For example, `{i,j} within workday` implies that `{i,j} within 8 hours`. While always correct, this approximation can be crude: an event received at 4pm will be stored for 8 hours instead of just 1 hour.

An alternative is to keep the general framework laid out in this article, but use a more advanced form of temporal reasoning. For example, one could allow not only numeric edge weights in the temporal distance graph but also symbolic ones like "workday." The notion and computation of the temporal distances (shortest paths) and the longest temporal distance (longest shortest path) must be adapted then. An important challenge is that calendrical notions are not generally comparable and summable like the numeric edge weights were, e.g., "workday" and "1 hour" cannot be compared or added easily. Research on temporal reasoning [13] will provide an extensive foundation for answering problems like this; note however that the particular problem that must be solved for temporal relevance (establishing temporal distances) is a not a standard problem.

We have made a further assumption that base relations which correspond to atomic event queries have only one occurrence time interval. It is a common request for complex event queries, e.g., to support several occurrence times given according to different clocks (important especially in distributed systems) or to distinguish a valid time and a detection time. The proposed algorithm can accommodate additional time stamps easily and can also deal with some types of relationships between these. For example when different clocks are synchronized with a precision of 1 second, this can be reflected by appropriate edges in the temporal distance graph. This addresses also in part issues related to a non-linear notion of time (e.g., only partially ordered), potentially requiring a combination with more advanced temporal reasoning (as above).

6.2 Dynamically Changing Query Plans

We have assumed the query plan to be static, i.e., the set of queries does not change at run time. Removing queries from a query plan at run-time is without problems — with removing the query, we also remove all its relations (if they are not needed by other queries) and the corresponding relevance conditions. Adding a new query can be problematic, however, depending on the semantics of adding a query.

In the simplest case, a new query that is added at a time point t_0 "sees" only events happening *after* t_0. Any result tuples that are generated using incoming events before t_0 are not considered part of the query's answer. This situation is fairly similar to the static case, and all we have to do for the new query is to determine its relevance conditions and add them to the existing relevance conditions.

In the most general case, a new query is supposed to see also all events that happened *before* its adding at time t_0. If there is no knowledge about the queries that can be added, no event can ever be deleted because it always might be needed by some query that is added in the future. This makes all events relevant, and there is no real need for relevance conditions.[6] It should be noted that this general case can be considered unsolvable because it implies maintaining a history of events that grows at least linearly with the length of the (potentially infinite!) event stream.

We can also consider an in-between case where a new query can see events that happened *before* it adding at time t_0, but the query is restricted some way. The restriction can be, e.g., that only some specific events during a finite time windows $[t_0 - w, t_0]$ will be relevant for answering it. This restriction could be included into the static determination of relevance conditions by including it into the original query plan, e.g., as a kind of "artificial query" that simulates the restrictions on queries that can be added in the future. The artificial query then expresses a kind of "worst case" for any future queries.

In another in-between case, we can use our method for determine whether adding a given new query (which is supposed to see events before its t_0) is possible. For this we would determine the relevance conditions of the new query Q and then compare it with the existing relevance conditions to see if all required event are available.

Note that the issues mentioned here are inherent to the semantics and evaluation of event queries and exist independently of any method for determining temporal relevance.

6.3 Joins in Temporal Relevance Conditions

Our temporal relevance conditions use only comparisons

[6]Even in such a scenario, relevance conditions can still be interesting though, e.g., if only those events that are needed for the current queries should be kept in main memory while a history of all events is maintained in secondary memory.

of the form $i > now - t$ or $j \geq now - t$. More expressive temporal relevance conditions, e.g., containing joins with other relations, could in some cases allow to remove more events. Consider $TR_{C_c \text{ in } F_f}$ of query plan 1 (Figures 1 and 3). A tuple $r \in C_c$ that is older than 4 hours should already have a corresponding tuple $r' \in D_d$ that will join with it. The selection $\sigma[\max\{c.e, d.e\} - \min\{c.s, d.s\} \leq 4]$ will eliminate join results produced with D_d-tuples arriving after more than 4 hours of (the starting time of) this $r \in C_c$. The temporal relevance condition for C_c could therefore be stronger, e.g., using a strawman notation akin to nested queries:

$$\begin{aligned} &TR'_{C_c \text{ in } F_f} \equiv c.s \geq now - 5 \wedge c.e \geq now - 5 \wedge \\ &\quad (c.s < now - 4 \rightarrow x \in \pi[x](\sigma[d.s \geq now - 4](D_d)) \end{aligned}$$

The nested query of "$x \in \ldots$" in the condition leads to a (semi-)join with relation D_d in the evaluation.

Such a join in relevance conditions is rather undesirable. It is expensive to perform, in particular since the intermediate result is not materialized and used in the query evaluation. We can achieve a similar effect, however, by using a different query plan that materializes the join of C_c and D_d. An example is query plan 2 of Figure 1; its $TR_{V_{c,d} \text{ in } F_f}$ has the same effect as the join of the above $TR'_{C_c \text{ in } F_f}$. As an additional benefit, the join is materialized and thus used in the query evaluation. Testing the relevance condition therefore does not add any overhead.

Note that the choice which query plan to use, i.e., in particular which intermediate results to materialize, is the responsibility of the query planner. Relevance times are an important input for its cost metric (cf. Section 1), and the cost metric should take into account not only the cost of producing query results but also of garbage collection, i.e., of evaluating temporal relevance conditions and removing tuples from their stores.

6.4 General Relevance

The (temporal) relevance conditions derived in this paper rely only on temporal conditions and we have made no assumptions on the contents of event streams (in particular their unknown future content). There are also other ways to determine the relevance of event tuples than temporal conditions; we call this relevance also "general relevance" to contrast it with (purely) temporal relevance.

On particularly interesting problem of determining relevance is using axiomatic knowledge about the future content of event streams; we also call this "axiomatic relevance." Such axiomatic knowledge can be stated explicitly as rules or derived implicitly from other sources such as business process descriptions. Axioms about event streams are similar to integrity constraints in databases in that they limit possible contents of the event streams. They are however differently used: In databases integrity constraints are *checked* to avoid or react to updates that violate them. In determining axiomatic relevance, the axioms are in contrast *assumed to be true* without checking them.

A vivid example of axiomatic knowledge about event streams are constraints on key references between event streams, which are natural in many applications: in shipping applications, a "shipping notice" event must be followed by either a single "delivery" event or a single "return to sender" event; the tracking identifier acts as a key that correlates these events. "Shipping notice" event tuples become irrelevant (and can be removed) as soon as they have found their unique join partner; without the constraint, the event tuple would have to be stored in case further join partners arrive. In [5], it is discussed how such a constraint speeds up query evaluation in data streams. Note however that there, the constraints are dynamically "mined" from the data streams not a priori specified as axioms.

7. RELATED WORK

To the best of our knowledge, the problem of determining temporal relevance introduced and solved in this article has not been considered before. In part, this can be attributed to characteristics of earlier languages for complex event queries, which are restricted in comparison to the rule language (cf. Section 2.1) used in this work.

One flavor of languages, popular especially in active databases research, builds complex event queries from atomic event queries and composition operators. These languages are commonly called event algebras. Composition operators can be understood as functions whose input and output are streams of events. For example, binary composition operators such as conjunction (often written $A \wedge B$) or sequence (often written $A; B$) take as input two event streams and produce as output another event stream containing the complex events. This event stream can in turn be combined with other event streams using further composition operators.

Many event algebras do not have temporal restrictions that would allow garbage collection through temporal relevance (e.g., [17, 16, 12, 1]). In part, this is due to the fact that event queries in these works are assumed to run inside (short-lived) database transactions. Outside of transactions, however, bounds on the "life-spans" of events and garbage collection become important [10]. Some event algebras therefore allow to add a temporal window to an operator or a subexpression (e.g., [22, 19, 11, 9]), e.g., $(A; B)_{1h}$ to indicate that B must follow A within 1 hour. The life-spans of [2] are also similar to this. Since the time window is applied to the full subexpression, determining temporal relevance of the involved events in trivial. However, this specification of a time window is very limiting, and many temporal conditions —like those of the query for `F(x)` in Section 2.1— cannot be expressed with it.[7] Further, rewriting such event algebra expressions for the purpose of query optimization is far more constrained [11] than for a relational algebra variant like CERA.

Query languages for (relational) data streams, e.g., CQL [4], typically require users to explicitly specify a time window with each event stream. For base relations, such time windows loosely correspond to temporal relevance conditions (although they usually are only for one time stamp). The main difference is that in our work, temporal relevance conditions are derived automatically from queries, while in data stream query languages temporal windows must be explicitly specified and affect query semantics. Note that data stream query languages usually have only very limited support for temporal conditions beyond the specification of such time windows.

Not really a dedicated event query language, but still a popular way to implement complex event queries are production rules. There, events (and states) are asserted as

[7]Note that $((C; D)_4; E)_1$ does not express the conditions of the query under interval-based semantics since the outer conditions entails that C and E happen within 1 hour not as required D and E.

facts and production rules then derive and assert new facts, e.g., complex events, or retract facts. Garbage collection, that is, retracting facts of events that have become irrelevant, must be programmed manually, e.g., by writing appropriate rules. Programming errors in this garbage collection are easy to make and will cause incorrectness of query results or memory leaks. An automatic garbage collection, as enabled by temporal relevance investigated in this work, is clearly preferable.

The following addresses more query plans with materialization points and their incremental evaluation than temporal relevance, but might still be of interest in the larger context, esp. to shed light on the relationship with production rules. Query plans with materialization points and their incremental evaluation (Sections 2.3 and 2.4) can be understood as a generalization of rete networks [14] in the following sense: rete always materializes the (intermediate) results of binary joins, while our query plans can choose their materialization points more freely, e.g., also materialize joins of higher arity. In this respect it is therefore also a generalization of TREAT [20], which in contrast to rete does not materialize any intermediate join results. However our approach is also very much a restriction of rete and TREAT in that in our incremental evaluation we only add tuples to materialization points. Deletion due to the retraction of facts, which is necessary in production rule systems, is not needed for our event query evaluation. We only need deletion as a garbage collection mechanism that does not influence query semantics.

8. CONCLUSION

In this article, we have introduced the problem of determining relevance of events and intermediate results for complex event queries, which is important for both garbage collection during query evaluation and for query planning. We have detailed the specific issue of temporal relevance and presented a method for determining it statically, i.e., at query compile time. Our method uses a so-called temporal distance graph, and the problem of determining relevance times reduces to finding the longest shortest paths in the graph. While our method is specific for query plans with materialization points using the CERA variant of relational algebra, our ideas, in particular the notion of the temporal distance graph, apply also to other formalisms and it should be easily possible to adapted them.

Relevance of events and intermediate results is, to the best of our knowledge, an issue given little consideration so far. As discussed in our outlook, we believe that there are many more aspects to it than just temporal relevance and that they deserve further investigation.

9. REFERENCES

[1] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data and Knowledge Eng.*, 59(1), 2006.

[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] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[5] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.

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

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

[8] F. Bry and M. Eckert. Towards formal foundations of event queries and rules. In *Proc. Int. Workshop on Event-Driven Architecture, Processing and Systems*, 2007.

[9] F. Bry, M. Eckert, and P.-L. Pătrânjan. Querying composite events for reactivity on the Web. In *Proc. Int. Workshop on XML Research and Applications*, 2006.

[10] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an integrated active OODBMS: Requirements, architecture, and design decisions. In *Proc. Int. Conf. on Data Engineering*, 1995.

[11] J. Carlson and B. Lisper. An event detection algebra for reactive systems. In *Proc. ACM Int. Conf. On Embedded Software*, pages 147–154, 2004.

[12] 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.

[13] M. Fisher, D. Gabbay, and L. Vila, editors. *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier, 2005.

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

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

[16] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. Int. Workshop on Rules in Database Systems*, 1993.

[17] 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.

[18] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1995.

[19] A. Hinze and A. Voisard. A parameterized algebra for event notification services. In *Proc. Int. Symp. on Temporal Representation and Reasoning*, 2002.

[20] D. P. Miranker. TREAT: A better match algorithm for AI production system matching. In *Proc. AAAI Natl. Conf. on Artificial Intelligence*, 1987.

[21] R. Sedgewick. *Algorithms in C*. Addison Wesley, 1990.

[22] D. Zhu and A. S. Sethi. SEL, a new event pattern specification language for event correlation. In *Proc. Int. Conf. on Computer Communications and Networks*, 2001.