Application of a Temporal Database Framework for Processing Event Queries

by

Foruhar Ali Shiva

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved November 2012 by the
Graduate Supervisory Committee:

Susan Urban, Co-Chair
Yi Chen, Co-Chair
Hasan Davulcu
Hessam Sarjoughian

ARIZONA STATE UNIVERSITY

December 2012

ABSTRACT

This dissertation presents the Temporal Event Query Language (TEQL), a new language for querying event streams. Event Stream Processing enables online querying of streams of events to extract relevant data in a timely manner. TEQL enables querying of interval-based event streams using temporal database operators. Temporal databases and temporal query languages have been a subject of research for more than 30 years and are a natural fit for expressing queries that involve a temporal dimension. However, operators developed in this context cannot be directly applied to event streams. The research extends a preexisting relational framework for event stream processing to support temporal queries. The language features and formal semantic extensions to extend the relational framework are identified. The extended framework supports continuous, step-wise evaluation of temporal queries. The incremental evaluation of TEQL operators is formalized to avoid re-computation of previous results. The research includes the development of a prototype that supports the integrated event and temporal query processing framework, with support for incremental evaluation and materialization of intermediate results. TEQL enables reporting temporal data in the output, direct specification of conditions over timestamps, and specification of temporal relational operators. Through the integration of temporal database operators with event languages, a new class of temporal queries is made possible for querying event streams. New features include semantic aggregation, extraction of temporal patterns using set operators, and a more accurate specification of event co-occurrence.

ACKNOWLEDGMENTS

I would like to thank the members of my committee: Dr. Susan Urban, Dr. Yi Chen, Dr. Hessam Sarjoughian and Dr. Hasan Davulcu. I am especially grateful for Professor Urban's continued guidance and dedication to seeing this dissertation through with me, even after leaving ASU.

I had the benefit of occasional discussions with my cousin, Dr. Raza Hashim, and I am grateful for his insights.

I would like to thank the Al-Mahdi community of Phoenix, where I had the privilege of making many good friends, who have been my family away from family.

I would like to thank my wife, Farnoosh, for her patience and support during this journey.

Finally, I would like to thank my parents. Without their support this would not have been possible.

TABLE OF CONTENTS

PAGE

CHAPTER PAGE

CHAPTER PAGE

CHAPTER PAGE

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

The increasing availability of network infrastructure and bandwidth has increased the prevalence of system designs in which autonomous entities share data through the use of events. An event represents a data notification pertaining to a specific instant in time. For example, in a smart home application, temperature readings or the entrance or exit of people from a room can be modeled as events. Homeland security, supply chain management, and any environment where sensor readings and event occurrences are monitored are examples of the many applications that require online processing of streaming data.

In event-driven applications, the direct communication of event data is fast, push-based, and, in general, happens when an entity decides it has new data to share. Based on the input, any recipient of the event might extract information that can be used to invoke actions or to generate new events. Extracting relevant information from these continuous streams of events is an important and relatively new challenge. The detection of events, and in particular, the detection of complex relationships among several different events, needs to happen in real-time. The simplest example of a temporal relationship between two events is that of a sequence, where one event occurs in time before the occurrence of another event.

The process of detecting complex events is similar to query processing over data streams. In contrast with traditional database queries, event stream queries add a temporal axis to existing data schema, which creates the need for languages that are enabled to query temporal data. Two distinct but related approaches are being pursued for querying this streaming data: Complex Event Processing (CEP) (Barga and Caituiro-Monge 2006; Eckert 2008; Wu, Diao, and Rizvi 2006; Cugola and Margara 2010; Gyllstrom et al.

2006) and Data Stream Processing (Patroumpas and Sellis 2006; A. Arasu, Babu, and Widom 2006). CEP typically focuses on detecting patterns in the order of occurrence of the incoming events. DSP, in contrast, focuses on filtering and aggregation of stream data. Another term that is found in the literature is Event Stream Processing (ESP). ESP is often used analogously to Complex Event Processing in the context of streams of events arriving over the network.

Temporal databases and temporal query languages (Date, Darwen, and Lorentzos 2002; Snodgrass 1995; Etzion, Jajodia, and Sripada 1998) have been a subject of research for more than 30 years, but have yet to gain widespread commercialization. Work in the area of temporal databases adds a temporal dimension to stored data, with query techniques for retrieval and comparison of temporal data. The use of temporal database query languages, however, has not yet been explored for ESP.

1.1 Comparison of Event Stream Processing and Temporal Database Concepts

Most of the earlier work in event processing considers events to be instantaneous and subsequently assigns a single timestamp value to each event. However, it has been demonstrated that point-based semantics cannot always correctly capture desired semantics in different applications (Adaikkalavan and Chakravarthy 2005). Additionally, point-based events do not always lend themselves well to intuitive operator semantics.

These issues have led to the increasing adoption of an interval-based model in the ESP research community. The interval-based model assigns a begin time and end time to each event. The interval-based approach does not suffer from the same semantic issues as the point-based approach, and it also supports sequence semantics in a way that upholds transitivity, which is intuitively expected.

Interest in temporal databases stems from the desire to assign a validity interval to relational data. While there are simple ways for a relational database to assign a time

interval to tuples, querying this temporal data is not always straightforward. The field of temporal databases has emerged primarily to deal with these challenges. In essence, temporal databases are databases that operate on temporal data. Their expressivity is analogous to that of relational databases with support for correlation of temporal data. Unlike ESP, temporal database queries are not specifically concerned with continuous processing of events that arrive over a stream of data. Temporal databases are typically used to query data that has a temporal dimension, which is sometimes referred to as historic data. For instance, the work in (Date, Darwen, and Lorentzos 2002) presents an example where a query requires periods of time during which a supplier supplied some part in a specific city.

Current work in stream and event processing languages imposes limitations on the expressivity of the query language in different ways. Stream processing systems typically support an SQL-like language to query the data. To cope with unbounded input streams, stream processing languages require the extensive use of windowing to avoid running into unbounded wait times and memory requirements. An example of a classic stream processing language is the Continuous Query Language (CQL) (A. Arasu, Babu, and Widom 2006). Several other languages employ the same concepts introduced in CQL (Purich 2010; Morrell and Stevan D. 2007; EsperTech). Event processing languages typically follow a composition operator-based model. Such languages offer a set of event operators, such as sequence or conjunction, which detect how the operand events are temporally related. The output of each operator is also an event.

A more recent approach to querying event streams can be found in XChange^{EQ} (Bry and Eckert 2007). In contrast to composition operator-based languages, the XChange^{EQ} approach treats detection of temporal relationships and event composition as two separate query dimensions. This separation of concerns results in more expressivity

and clearer semantics. But the semantics of what constitutes an event places strict schema and temporal restrictions on the query input and output.

In contrast, temporal databases don't impose any temporal conditions on the use of querying constructs. In fact, temporal databases can be viewed as a generalization of relational databases (Date, Darwen, and Lorentzos 2002) in which temporal versions of traditional relational database operators are supported. Traditional database operators can be viewed as a specialization of this model in which the temporal dimension is ignored.

The approach in (François Bry and Eckert 2007) performs complex event processing using operators that are implemented using a restricted version of relational algebra with constructs added for event composition. This is significant because it demonstrates a fundamental compatibility between relational database queries and complex event processing. However, the restrictions imposed do not allow the same level of temporal expressibility as that found in temporal databases.

1.2 Outline of Research Objectives

This research aims to bridge the gap between complex event processing and temporal databases by developing a framework that allows the use of temporal relational operators for querying event data. In particular, this research involves extending a relational framework with language features and architectural requirements for on-line processing of temporal event queries.

Different architectures have been used for efficient evaluation of event queries. Much of the work in the area of Active Databases and CEP systems use a Finite State Automata (FSA) or a similar structure for evaluating event queries (Gyllstrom et al. 2006; Brenna et al. 2007; Gatziu, Geppert, and Dittrich 1995). In these approaches, the FSA is an internal representation of the composite event definition. The FSA model enables sharing of intermediate results between different queries but cannot directly handle

concurrent occurrences of events or negation. The FSA approach is also rigid in terms of the order of operations performed. The use of Petri-Nets has also been proposed in a similar manner (Gatziu, Geppert, and Dittrich 1995).

Some of the more recent approaches in this area, such as XChange^{EQ}. (François Bry and Eckert 2007) and ZStream (Mei and Madden 2009), utilize a query processing approach. The query processing model, aside from being well understood, has the benefit of building on already established query optimization techniques. It also has the flexibility of reordering operations for optimization purposes. ZStream propagates events from event buffers through an internal tree structure when an evaluation cycle is triggered. Language-wise, it is a composition operator-based language with a mandatory window specification. The perceived limitation of data under consideration has led to the design decision of keeping track of all concerned events in all nodes of the evaluation tree.

In contrast, in the XChange^{EQ} model no mandatory temporal limitation is placed on the constituent events. The framework supports materialization of intermediate results which can be used for generating results more efficiently. Moreover the XChange^{EQ} model, allows the flexibility of choosing which intermediate results should be materialized.

Adoption of interval semantics in event processing languages has adequately dealt with semantic problems with the point-based approach. However, event operators originally defined in the point-based settings are not expressive enough to capture the complex queries that are possible in an interval-based setting. This research advocates the idea that operators developed in the context of temporal database query languages are a natural fit for querying interval-based event data and enable a new dimension for expressing event queries.

This dissertation presents an integrated approach for employing temporal relational operators for querying of event streams through a new event processing language called the Temporal Event Query Language (TEQL). TEQL enables reporting temporal data in the output, direct specification of conditions over timestamps, and specification of temporal relational operators. Database operators and, by consequence, temporal relational operators cannot be directly applied to event streams. This is due to the fact that the relational model includes non-monotonic and blocking behavior, which is undesirable in an event setting. In order to address this issue, a preexisting relational framework for querying of event streams, called CERA (Eckert 2008), is extended to support TEQL queries. Semantic extensions to CERA to support temporal operators are identified and formalized. The extended framework supports continuous, step-wise evaluation of temporal queries. The incremental evaluation of TEQL operators is formalized to avoid re-computation of previous results. Through the integration of temporal database operators with event languages, a new class of temporal queries is made possible for querying event streams. New features of TEQL include semantic aggregation, extraction of temporal patterns using set operators, and a more accurate specification of event co-occurrence. A prototype implementation has been developed to demonstrate the feasibility of the ideas presented in this dissertation.

1.3 Dissertation Organization

The rest of this dissertation is structured as follows. A detailed overview of the work done in the related areas of stream processing, event processing, and temporal databases is presented in Chapter 2. Temporal relational operators are introduced and their applicability in an event environment is discussed in Chapter 3. Chapter 4 introduces the language design for TEQL, explaining individual language operators. Chapter 5 discusses the relational framework used for specification of temporal event

queries and formalizes the TEQL operators. Chapter 6 discusses the incremental evaluation of TEQL operators. Chapter 7 describes the prototype implementation. Chapter 8 discusses the expressivity that is achieved through integrating temporal relational operators in an event language by way of comparison with operators present in earlier languages. Chapter 9 presents a summary of the research and outlines directions for future work.

Chapter 2

RELATED WORK

This chapter provides an overview of related work. Section 2.1 outlines event processing work that has originated from the area of active database systems. Section 2.2 addresses the work done in the area of distributed event systems. Section 2.3 outlines issues with approaches that utilize composition operators. Section 2.4 provides an outline of work done in stream processing languages, by presenting a detailed look at CQL (A. Arasu, Babu, and Widom 2006), a language that provides the basis for many commercial stream processing languages. Section 2.5 presents the more recent work done in event stream processing. Section 2.6 addresses research on the specification of event processing languages. Finally, Section 2.7 summarizes and provides a discussion of relevant issues with existing approaches as presented in the related work section.

2.1 Composite Events in Active Database Management Systems

Complex event languages find their roots in active database research (Chakravarthy and Mishra 1994). The necessity to react to different kinds of events occurring within a DBMS gave rise to active databases that use the Event-Condition-Action (ECA) format for rules. In active databases, the event component is used to describe what has occurred. The condition component describes any conditions that need to be fulfilled, in order for the specified action to be taken. The condition is only checked upon occurrence of the event. The event itself can be a composition of different event types, in which case it is called a composite event.

SNOOP (Chakravarthy and Mishra 1994) is an example of a language that supports composite events for active databases. SNOOP provides the following operators for composing events:

Sequence: Detects the sequential occurrence of events.

Conjunction: Detects the occurrence of several events, regardless of their relative order.

Disjunction: Detects the occurrence of one out of several specified events.

A-Periodic: Detects the occurrence of an event, between the occurrences of two other specified events. Some languages implement an n-ary sequence operator which can provide a similar functionality.

ANY m out of n: As the name suggests this operator is similar to a disjunction operator, but it requires that a specified value 'm' out of the 'n' specified event types occur.

Non-Occurrence or Negation: This operator requires that the specified event does not occur. Negation needs to be bounded by a specified event or an amount of time.

Chakravarthy et. al. (Chakravarthy and Mishra 1994) argue that composing all possible combinations of input events would lead to a large number of composite events being generated that are not of interest to the programmer. On the other hand, these unwanted events cause a lot of overhead on the system by requiring a complete history of all involved events to be retained and matched against incoming events at all times. As a result, *consumption modes* (also called parameter contexts in some work) were defined to modify the standard behavior of the complex event statement to which they are applied. However, the operators themselves are only formalized in the unrestricted context in which all combinations of events available in the event history are considered.
As described in (Chakravarthy and Mishra 1994), the initiator and terminator of a complex event are defined as follows:

- The initiator of a complex event is the event that begins the process of detecting a composite event

- The terminator of a complex event is the event that ends the process of detecting a composite event and possibly generating an occurrence of the complex event.

The consumption modes introduced consist of the *recent*, *chronicle*, *continuous,* and *cumulative* modes. Consumption modes rely on the terminator of a composite event to decide which instances of constituent events are selected for generating the composite event. The recent and chronicle mode, cause the latest and earliest instances of the constituent events to be selected, respectively. With the cumulative mode, the parameters of all instances are accumulated to support aggregation of event parameters. The continuous mode detects a complex event over all possible combinations of event instances.

SNOOP combines the notions of event selection and event consumption into a single dimension. This limits the language to specific predefined combinations of selection and consumption policies. Additionally, in SNOOP, consumption modes are associated with composite event statements and not the constituent event types. As a result, a sub-expression of a composite event is a different kind of entity than the composite event statement since a consumption mode cannot be associated with a sub-expression. This creates a non-uniform language composition and destroys the benefits of equivalence between sub-expressions and named composite events.

ODE (Gehani, Jagadish, and Shmueli 1992) is an active object-oriented database system that supports triggers and constraints. ODE combines the Event and Condition parts of an ECA rule, presenting an Event-Action model. The basic events are events that are found in a traditional database setting such as start and end of a transaction. Timer events including relative timers are also supported in ODE. ODE also supports specification of composite events utilizing operators such as conjunction, disjunction, sequence, negation and selecting the n^{th} event in a sequence of events. The event model

assumes instantaneous events. The event specification follows a regular expression approach and it is implemented using an automata model.

SAMOS (Swiss Active Mechanism-Based Object-Oriented Database System) (Gatziu, Geppert, and Dittrich 1995) is an active object-oriented database. In SAMOS, the primitive events can originate based on the database or be user-defined timer events. SAMOS supports a rule definition language to specify ECA rules. SAMOS attempts to simplify event specification by supporting a limited number of orthogonal event operators. These include the usual conjunction, disjunction, and sequence operators. In addition, a * and a last operator are used to signify the first time and last time an event type occurs. A history operator, TIMES(n,E), is used to detect that E occurs n times. The *, last, negation, and TIMES operators all need to be specified within a time window. The time window is specified using a begin and end time which can be the occurrence of an event, or relative or absolute event times. The * and last operators can be considered simpler versions of what was later developed into consumption modes.

2.2 Events in Distributed Systems

The roots of complex event processing can also be traced back to distributed event based systems (Mühl, Fiege, and Pietzuch 2006). This is actually the field where the term "complex event processing" originates, in contrast to database systems where the term "composite event detection" was originally used. The main thrust of this line of research has been to provide a software layer that facilitates communication between the application and network layers in distributed applications for developers. This new layer, called the middleware, is utilized in large distributed systems (Luckham 2007) and implements a publish/subscribe model to support communication between different components. In this model, the publisher generates and submits event notifications. The

subscribers can subscribe to events that interest them. The middleware is in charge of filtering the generated events and delivering them to appropriate subscribers.

Much of the focus of this area of research has been on efficient filtering of events with different filtering algorithms being developed targeting different distribution of components and different topologies (Bittner and Hinze 2004). A significant amount of the work in the field of distributed event-based systems focuses on subscriptions that specify topics in the form of hierarchies or allow specification of simple value filters on the notifications (Mühl, Fiege, and Pietzuch 2006). However, most of the work in this area does not address the correlation of events or specifications of temporal relations between events, which is a topic of primary interest to this research. These issues are primarily addressed in some of the more recent work (Pietzuch, Shand, and Bacon 2004; Sánchez et al. 2003).

Similar to languages found in the database area, the Event Correlation Language (ECL) described in (Sánchez et al. 2003) is based on composition operators and follows an automata based model. A couple of languages in this area also support negation (Dong Zhu and Sethi 2001; Hinze and Voisard 2002). In these languages negation needs to be specified together with a time window. The work in SEL (Dong Zhu and Sethi 2001) supports several kinds of fixed and sliding windows, with the option to have one end of the window fixed and the other sliding.

In general the work in this area is more concerned with efficient routing of subscriptions to the subscribers and reducing network load, and besides a few notable exceptions, does not focus on composite events or temporal conditions. As seen above, when these issues are addressed, the approaches are similar to work found in active databases.

2.3 Issues with Composition Operator-Based Approaches

As mentioned in the previous sections, using operators that compose simpler events into more complex events has been a very popular approach for languages that are used to process events in different settings. These languages are sometimes collectivity referred to as event algebras, which reflects the fact that they produce new events by combining different events using the provided operators.

The composition operator-based approach for specifying patterns of events seems intuitive at first glance. However, existing implementations of this approach suffer from several problems, among them disagreeing and often ambiguous semantics across different languages (Zimmer and Unland 1999). Semantic issues with the composition operator-based approach have been presented in previous work, for example, in (White et al. 2007). A good overview can be found in the literature review and chapter 17 of (Eckert 2008).

The operators commonly found in these languages include sequence (;), conjunction ($\wedge$), disjunction ($\vee$) and negation. Several languages add more operators, but the added operators are not always orthogonal to these common operators. Additionally, some languages support sequence, conjunction, and disjunction in binary format, while others introduce n-ary versions of them which can take more than two events as operands. In the remainder of this section an overview of semantic issues with composition operators is presented:

The sequence operator: The concept behind the sequence operator is straightforward; it is supposed to signify when two events of specified types happen after each other. However, existing implementations offer different semantics for the sequence operator. Some languages require that for a sequence operator to be detected, no other event should occur between the two events participating in the sequence, while other languages drop

this requirement. Another issue with the sequence operator more specifically applies to languages that do not support interval-based semantics. Intuitively it would seem that the sequence operator should be associative. For example, the event pattern A;(B;C) should be equivalent to (A;B);C. In practice however this is not the case (White et al. 2007). In the first pattern when B;C is evaluated the timestamp of C is assigned to the sequence and the occurrence time of B is lost. As a result no restriction is enforceable on the order of A and B. In other words the first expression above is really equivalent to B;(A;C), which besides being entirely unintuitive, is clearly not equivalent to the second expression above.

Finally, it should be noted that when interval semantics are employed, the relationship between two events is no longer restricted to one happening after the other. Allen (Allen 1983) identifies 13 different relationships that are possible between two intervals: before, contains, overlaps, meets, starts, finishes, equals, and their inverses after, during, overlapped by, met by, started by, finished by. Equals does not have an inverse. The work in (Zaidi 1999) further extends these relationships by distinguishing between time points and intervals and the relationships they can have with each other. As is noted in (Eckert 2008), there is an important difference between these relationships and composition operators. A relationship merely maps to a true or false value based on the intervals that are submitted to it. However an operator takes two events as input and generates an output stream of events. The language in (Roncancio 1999) proposes operators based on these relationships. Besides the sequence operator, which maps to the before relationship, there is usually no n-ary extension to binary operators based on Allen's relationships. As will be discussed later, specifying temporal distance can be problematic with binary operators.

The conjunction operator: The conjunction of two events occurs when an instance of each one occurs, regardless of which one happened first. While the symbol for conjunction in event processing is borrowed from logic, it is important to note that the interpretation of conjunction is different in event processing than its counterpart in logic. Consider the following example:

(A∧B);C

The logical interpretation dictates that A and B both need to occur before C. However this statement cannot be rewritten as:

(A;C)∧(B;C)

While logically the two statements seem to be saying the same thing, the second statement allows for two different instances of C to be used to trigger the event pattern. For instance the pattern of events a, c1 , b, c2 causes the first statement to be fired once using c2, while the first statement will fire twice, first using c1 in conjunction with a and then c2 (Eckert 2008).

In general, reusing the same event type more than once in an event pattern can lead to confusion. One work around could be to enable the user to specify aliases for the event types being used, thus signifying whether an event type that is referenced more than once needs to always refer to the same instance of the event or is bound to instances independently of other references.

The disjunction operator: The disjunction operator fires when one or the other of its operands occurs. In general, the semantics of disjunction are straightforward. However, there are different options on what can happen when more than a single operand occurs. Consider the following example from (Eckert 2008):

A; (B∨C);D

If both b and c occur between a and d, the entire event sequence might be triggered once or twice depending on how the semantics of the language is defined.

Negation: Negation signifies that a specified event has not occurred. Since the incoming stream of events is unbounded, a temporal bound needs to be specified, after which the non-occurrence of an event can be signaled. (Chakravarthy and Mishra 1994) disagree with treating Negation as an operator. Since it's semantics do not conform with the basic requirement of something happening at a specific time. Alternatively, negation is sometimes proposed as a separate condition instead of an operator to avoid this theoretical objection.

Specification of Temporal Distance: Finally, many languages couple composition operators with the specification of a temporal distance. This allows the user to specify not only the order of events participating in an operator but also how far they are allowed to be from each other. For example $(A;B)_{1h}$ specifies that an instance of B needs to happen after an instance of A but within 1 hour from it. This approach seems straightforward, but has some unexpected restrictions as demonstrated in (Eckert 2008). For example, assuming interval-based semantics, if we want to detect a pattern in which A happens and then B happens within an hour of A. If C must happen within an hour of that same B, this query cannot be expressed in most existing languages. The two candidate expressions for specifying this query are:

$((A;B)_{1h};C)_{1h}$

$(A; (B;C)_{1h})_{1h}$

It should be noted that both of the above queries require that C occur within an hour of A, a requirement which was not present in the original query.

This concludes the overview of issues in composition operator-based languages. Essentially, composition operators mix two separate query dimensions together. For

instance, the sequence operator simultaneously composes two different events together while imposing a temporal condition over them. The approach of this research is similar to that of XChange^{EQ} in which event composition and temporal conditions need to be specified separately. This approach also avoids issues discussed above. The adopted language design is discussed in Chapter 3. The following section addresses the related work in the areas of stream processing languages.

2.4 Stream Processing Languages

Stream Processing Languages aim to query data that is arriving over incoming streams. The term stream is used to typically signify data that is continuously arriving over a network medium. In this setting, the input streams have a very high throughput and so it is undesirable or even infeasible to store the incoming data before processing it. Stream processing languages themselves tend to resemble SQL. Some of the more prominent stream processing research projects include (Abadi et al. 2003; A. Arasu, Babu, and Widom 2006).

The requirement to process queries without the ability to indefinitely store the data, coupled with the fact that the input streams are theoretically unbounded, raises the necessity to limit the number of elements under consideration. This necessity is met by specifying windows, which are sets of recent elements that are currently under consideration. The number of items in the window is limited by either a specific number of events, or based on a specified duration of time. There are also other kinds of windows that determine whether a tuple is allowed in the window or not, based on its data value. These kinds of windows are referred to as semantic windows (Qingchun Jiang, Adaikkalavan, and Chakravarthy 2007).

CQL (Continuous Query Language), developed at Stanford University (A. Arasu, Babu, and Widom 2006), formalizes some of the important concepts in Stream

Processing Languages and is the basis for several query languages used in commercial products (Purich 2010; Morrell and Stevan D. 2007; EsperTech). The following subsection elaborates on CQL as a representative example of stream processing languages and then outlines different forms of window semantics as described in (Patroumpas and Sellis 2006).

2.4.1 CQL

CQL considers windows as time-varying relations. A time-varying relation, R(t), at any point in time t, will include a set of tuples that conform to the window specification associated with it. The tuples in a time-varying relation do not persist in the relational sense. With the passage of time, new tuples are added to the relation, and older ones are removed from the relation based on the window specification, and without requiring an explicit data manipulation operation. CQL provides three classes of operators: Stream-to-Relation, Relation-to-Relation, and Relation-to-Stream. The original semantics for CQL lacked Stream-to-Stream operators, with the bulk of data manipulation performed using the Relation-to-Relation operations, which are equivalent to SQL operations.

Stream-to-Relation Operators:

Stream-to-Relation operators are window specifications. Three main kinds of sliding windows are supported: time-based, count-based, and partitioned windows. Sliding windows, in principle, add the newest items appearing in the input stream to the window, and remove from the oldest items already in the window. A time-based sliding window always contains the tuples that have appeared in a specified length of time from the current point in time. Count-based windows always contain the last n items that have appeared on the input stream. Partitioned windows first form sub-streams from the input stream based on the values of a specified set of attributes, similar to a Group By clause in SQL. Then the last n tuples from each group are retained in the window. There are also a

pair of special windows that are useful for formulating different queries. The Now window, always contain tuples that have arrived within the duration indicated by the current value of the timestamp. The Range-Unbounded window contains all items that have ever appeared on the stream.

Relation-to-Relation Operators:

These consist of standard SQL data manipulation operations, with the difference that they operate on time-varying relations instead of standard ones. The following example taken from (A. Arasu, Babu, and Widom 2006) demonstrates a stream-to-relation operation, and relation-to-relation operations.

```
select distinct vehicleId
from posSpeedStr [range 30 seconds]
```

A time-based window with the length of 30 seconds is first defined over the input stream posSpeedStr. The result is a time-varying relation. SQL-like operations can be performed over this time-varying relation. In the above example, vehicles with distinct identifiers are selected from the time-varying relation. This makes the result of the query another time-varying relation.

Relation-to-Stream Operators:

Relation-to-Stream operators take a time-varying relation as input and produce a stream as output. There are three Relation-to-Stream Operators:

- Istream, or Insert Stream - returns the stream of tuples as they are inserted into a relation.
- Dstream, or Delete Stream - returns the stream of tuples as they are removed from a relation.
- Rstream, or Relation Stream - if any item exists in R(t) it will also be in the stream with timestamp t.

The following example from (A. Arasu, Babu, and Widom 2006) demonstrates the use of the Insert Stream operator in combination with a range unbounded window:

```
select istream(*)
from posSpeedStr [range unbounded]
where speed > 65
```

First an unbounded window is defined over the input stream, which results in a time-varying relation that includes every tuple that appears in the stream. The filtering operation declared in the where clause omits tuples with speed less than 65 from the relation. Finally, the istream operator outputs every new tuple that is added to the time-varying-relation as a new stream.

As mentioned before, CQL in its original form lacks Stream-to-Stream operators. This implies that even for simple filtering of a stream, a window definition and subsequent transformation into a stream is necessary. Operators similar to event composition operators were not originally supported. Some of the later work in languages based on CQL, such as Oracle CQL (Purich 2010), support a limited number of direct stream-to-stream operations. Most significantly the MATCH_RECOGNIZE clause in Oracle CQL allows patterns of values, such as a W-pattern[1] in the value of some attribute to be detected.

CQL cannot be extended to support a continuous time model because time-varying relations and relation-to-stream operators require discrete time points. CQL does not support semantic or value-based windows which can be useful to group more meaningful data together. Tuples, while within a window, follow the relational standard of belonging to a set/bag. As members of a set/bag, tuples are no longer ordered. This implies that the relation-to-relation operators that are designed to undertake the bulk of

[1] A W-pattern means the following pattern of values in an attribute: decrease, increase, decrease and a final increase

data manipulation in CQL, cannot make use of any ordering information or detect a temporal pattern.

Istream and Dstream act purely on data values. If a new tuple with the same data values of a tuple that is about to be removed from the window arrives, the insertion of the new tuple and the deletion of the old one will be missed by Istream and Dstream, respectively. This could be rectified by defining other versions of Istream and Dstream that are sensitive to timestamps.

2.4.2 Window Semantics

Attempting to directly apply data manipulation operations from the database world to stream data is problematic. Given the high volume of stream data, it is usually not possible to have to store the entire stream history, and even if it were possible it may not be practical to have to examine the entire history of incoming streams to produce a new result. Even if we limit ourselves to monotonic queries that only append new results, there are various problems that can arise. Blocking operators such as aggregation are unable to produce a single result without access to the entire event history. Other operators that are stateful such as join or intersection also cause problems due to the fact that every newly arriving tuple needs to be matched against the entire event history. Patroumpas and Sellis formalize window semantics for continuous queries over data streams in (Patroumpas and Sellis 2006). The importance of windows in Stream Processing systems is due to the ever increasing volume of incoming data. Windows, in effect, limit the number of items that are considered for evaluating a query at any moment. This also makes evaluation of blocking operators feasible, because they will only need to consider items that are currently in the window, instead of waiting on an input stream that never ends. On the other hand, even for non-blocking operators, the number of items that need to be considered at any given instant is bounded by the

window. This becomes even more significant when the query requires a join-like operation that considers multiple input streams.

To address these issues, Patroumpas and Sellis offer a categorization of different kind of windows possible. First windows can be categorized into physical (or time-based), and logical (or count-based) windows. Additionally a window can be specified either by marking its two edges or by using an edge along with the desired size which can be either in logical or physical units. The window bounds can be fixed or variable, in which case a progression step can also be specified.

Different combinations of these conditions lead to different window types that are formalized using algebraic expressions in (Patroumpas and Sellis 2006). The main categorization of the windows is as follows:

Physical Windows: Physical windows are sliding by default and have the following sub-types:

Count-based Windows: A count-based window contains a fixed maximum number of the most recent tuples.

Partitioned Windows: Similar to a GROUP BY clause, a partitioned window divides an incoming stream into partitions that have the same value for a specified set of attributes. A count-based window is then applied to each substream using a specified value. The contents of the partition at any given instance, is the union of the contents of these partitioned windows.

Logical Windows: Inclusion of events in a logical window relies on the timestamp of the incoming tuple. The following main logical window sub-types are defined:

- Landmark Windows**:** A landmark window encompasses different kinds of windows that have one bound fixed, but the other bound is allowed to progress with time.
- Fixed Band Windows**:** A fixed-band window has both bounds fixed to specific points in time.
- Time Based Sliding Windows**:** In its basic form, a time-based sliding window contains the most recent items that occur within a specified amount of time.
- Time Based Tumbling Windows**:** Time-based tumbling windows rely on a hop unit, where the hop unit is equal to the window length. As a result no tuple is repeated in two consecutive states of the window. This can be useful when the tuples need to be aggregated in groups.

Patroumpas and Sellis additionally provide definitions for relational operations such as join, union and aggregation over these windows. Windows that may be defined over semantic conditions are not considered. An equivalent mechanism to consumption modes is also not offered.

2.5 Event Stream Processing

Event stream processing languages are used to detect temporal patterns of incoming events. The events themselves can be thought of as time-stamped tuples, with different event types having different headings. Events can be categorized into two kinds: primitive and complex events. Primitive events are events that are generated by the operational environment. Examples of these might be an operation in a database system, or a sensor or RFID reading. Complex events, on the other hand, are patterns of events detected by the event processing system. A complex event relies on several events occurring in a specified pattern. For example, a complex event may require that two primitive events happen concurrently or one after the other.

Event stream processing languages, while conceptually similar to event specification of ECA rules in active database systems, go beyond basic event patterns. ESP languages typically add features for condition filtering that are traditionally found in the condition part of an ECA rule. ESP languages also offer composition operators that include sequence, conjunction, disjunction, repetition, and the a-periodic operator as defined in (Chakravarthy and Mishra 1994).

The result of each operator is then a composite or complex event, which can then be input to other event operators to form even more complex events. Event processing languages typically do not support a window structure similar to that found in DSP Languages.

In terms of their architecture, composition operator-based languages usually use an automata or a similar structure internally to evaluate event patterns. While these kinds of structures are helpful for optimizations across multiple queries, unlike query plans, they are rigid in terms of their evaluation order. Also, it is not always straightforward to handle simultaneous occurrence of events using an automata structure.

The languages in (François Bry and Eckert 2007; Mei and Madden 2009; Barga and Caituiro-Monge 2006), use a query plan based approach, which offers flexibility in terms of order of query operations similar to optimization of database queries. This leads to optimization opportunities in terms of space utilization for intermediate results that are materialized.

In the following sub-sections, an overview of some of the prominent event stream processing languages is provided.

2.5.1 SASE

SASE (Gyllstrom et al. 2006) is one of the first projects to utilize a query-plan based approach for event processing. The objective of SASE is filtering, correlation, and

detection of complex event patterns over RFID streams of data. Similar to relational databases, SASE builds a query plan in the form of a tree structure, with the different query operators forming the nodes of the tree.

To leverage the fact that the data arrives over time and the queries are looking for sequences of data, SASE uses a special operation called Sequence Scan and Construction (SSC). SASE first extracts the specific sequence of events that are required from the query, ignoring negations. Checking for this sequence over incoming streams is the job of the SSC operator, which forms the basis for all query plans. SSC itself is implemented using an automata mechanism. SASE further optimizes the detection of event sequences by pushing predicates and time window conditions down in the query plan.

When composing events, SASE includes the entire heading of the contributing event. When only a subset of data is necessary for the resulting event, this causes an excessive amount of overhead. The situation can become worse if events with large headings or a correlation of numerous events types are being considered. Ultimately, SASE relies on composition operators; and shares some of the same semantic issues composition operator based languages. SASE does not support the use of complex events when specifying other complex events. In SASE, events are instantaneous. Interval-based semantics and aggregation operators are not supported.

2.5.2 MavEStream

MavESstream (Qingchun Jiang, Adaikkalavan, and Chakravarthy 2007) aims to integrate the event and stream processing models. The proposed model uses a three layer approach. At the lowest layer, a stream processing engine handles incoming streams and performs filtering and aggregation operations. At the next layer event processing occurs over filtered streams that can be treated as input events. Finally, the third layer is a rule

processing layer that can trigger rules based on incoming events. Event specification takes the following form in MavEStream:

CREATE EVENT *Ename*
SELECT A_1, A_2, ..., A_n
MASK *Conditions*
FROM *ES* | *EX*

where Ename is a new event being defined and A_1, A_2, ..., A_n are attributes. A mask is an attribute-based constraint that can be pushed down to sources. ES and EX in the from clause represent the data sources that can be continuous stream queries or event pattern specifications.

A notable concept introduced in MavEStream is that of a *semantic window*. A semantic window is distinct from a count-based or time-based windows in that the window items are determined by a user-specified condition. The condition itself can involve the parameter values of the events involved. While a semantic window seems to be a valuable tool, MavEstream's approach to implementing it does not seem very intuitive: the user-specified condition is iteratively checked by removing older items in the window. No checking is done on entry of new items into the window. This mechanism is rather inflexible: it enables simulation of time-based and count-based windows, but lack of control over events entering the window and strict, in-order, removal of existing items make the mechanism rigid and only appropriate for very specific applications.

2.5.3 CEDR

CEDR's (Complex Event Detection and Response) stated goal is to attempt to unify three different approaches to temporal data arriving over streams (Barga et al. 2006), namely: event processing, stream processing and publish/subscribe systems. The authors argue that the main difference between the different models is the workload that is expected of the target systems. In order to unify the three different models, different consistency

levels are supported. A consistency level is based on a target system's tolerance of out of order processing of events. In order to achieve unification, the consistency level can be directly specified by the user. The entire stream is viewed as a time-varying relation, with each event playing the role of a tuple in the relation. All tuples also need to have a unique ID.

Three different timestamps are supported for achieving the diverse requirements of the underlying systems. The first dimension is the *validity interval* assigned by the provider of the event. The ID allows an event provider to reference an event which it has 'inserted' earlier and to change its validity interval or other attributes. The second time interval, called *occurrence time,* indicates when modifications are made by the provider. Conceptually, this second interval is comparable to transaction time in temporal databases (see Section 2.7). Tuples with expired occurrence times are no longer valid from the point of view of the event provider and have been replaced with a tuple with the same ID and a currently valid occurrence time. This can be viewed as a modification operation from the relational database perspective.

The following CEDR event example is from (Barga et al. 2006):

```
EVENT CIDR07_Example
WHEN UNLESS(SEQUENCE(INSTALL x,SHUTDOWN AS y, 12 hours),RESTART AS z,
5 minutes)
WHERE {x.Machine_Id = y.Machine_Id} AND {x.Machine_Id = z.Machine_Id}
```

The language of CEDR uses a composition operator-based language. The WHEN clause is used to specify the event operators. Multi-ary operators to detect sequence and m out of n input events are supported. Negation is supported using multiple operators. In the above example the UNLESS operator is an example of a negation. The UNLESS operator, and by consequence the entire event, is triggered if the sequence is not followed by a 'Restart' event within 5 minutes. As can be seen, the sequence operator is also

bound by a 12 hour time window. A mandatory time window is required for all event operators that correlate different input events.

In (Barga et al. 2006), it is said that event selection and consumption are supported, and the observation is made that selection and consumption need to be decoupled from operators and associated with input streams. This is a welcome observation and is likely to reduce the possibility of confused semantics sometimes found in other implementations of consumption modes. However, selection and consumption semantics are not presented nor are any examples provided.

The query evaluation is performed using pipelined operators that form a query plan. The third dimension, called CEDR time is the clock of the server and is used for handling out of order events. CEDR time cannot be queried by the language. By looking at CEDR time and the timestamp of the provider, it can be determined whether an event has arrived out of order or not. CEDR time is additionally used to correct incorrect outputs produced based on out of order events. The CEDR time of an output that is determined to be incorrect is terminated, and the correct output is produced with valid CEDR time assigned to it.

CEDR falls into the family of composition operator-based languages. CEDR allows the query language to define windows over the desired validity intervals or occurrence times for any given query. But manipulating timestamps is otherwise limited to the use of composition operators, which are limited in their expressivity. Materialization of different intermediate results and its effect on performance are not explored.

2.5.4 Cayuga

The Cayuga project (Demers et al. 2005) aims to develop an expressive stream processing language by extending event processing languages. The emphasis is on

achieving high performance and scalability. In order to achieve this, an event algebra named CESAR (Composite Event Stream AlgebRa) is developed. CESAR is evaluated using an automata-based approach, which enables cross query optimizations, similar to those found in some publish/subscribe systems, e.g. (Diao et al. 2002). As the name implies, CESAR uses event composition operators. The extensions to publish/subscribe event languages include support for value correlation and aggregation of events.

CESAR supports time intervals. The basic binary operators supported by CESAR are Union and Sequence. There is also an Iteration operator that can be used for checking for monotonic increase in a parameter value by repeated application of the sequence operator.

2.5.6 XChange^{EQ}

XChange^{EQ} is a more recent language that was designed for querying events. It is significant because it introduces a new style for querying events which is different from both composition-operator based event languages and data stream languages. The style introduced is similar to logical formulas, but the language is more user friendly and targeted specifically for events (Eckert 2008; François Bry and Eckert 2007). In this section, we present an overview of the language design in XChange^{EQ} and discuss some of the areas in which it is deficient.

Bry and Eckert (François Bry and Eckert 2007) argue that different querying dimensions have traditionally been mixed in event processing languages. They identify four separate dimensions for an event query language. They are:

- **Data Extraction:** Refers to the data that needs to be extracted from input events. This is done through variable bindings. The data is then used for testing query conditions, comparing and combining with persistent data, constructing new events, or for triggering actions.

- **Event Composition:** This dimension refers to the constructs that allow different events to be juxtaposed and create complex events. The composition constructs are sensitive to data, allowing the correlation to be limited to equivalent data across the events.
- **Temporal relationships:** Refers to different query conditions that involve time. XChange^{EQ} breaks temporal relationships into two kinds: qualitative and quantitative. Qualitative relationships are concerned with the ordering of different events while quantitative ones are concerned with the amount of time elapsed between their occurrences.
- **Event Accumulation:** Aggregating and checking for non-occurrence of events over an infinite input stream require setting a bound on the number of events being considered. Event accumulation is the dimension that deals with this issue.

According to these criteria, composition operators, which can be found in previous work, such as the sequence operator involves two different dimensions. Since two (or more) events are used to produce a new one, a sequence operator involves the event composition dimension. Furthermore, since the two events need to be ordered in a certain way, the sequence operator also enforces a temporal relationship. Eckert and Bry argue that this is undesirable because it causes mixing of inherently separate concerns. As a result, in XChange^{EQ} the sequence operator is not included. The only composition operators supported in XChange^{EQ} are conjunction and disjunction. The qualitative, and quantitative temporal relationships are expressed in the WHERE clause of an event query statement, with constraints on event time stamps.

The temporal model used in XChange^{EQ} uses time intervals to designate the occurrence time of events. For simple/primitive events, the start time and end time of the interval are assumed to be the same. For complex events, the time interval consists of the

time period that subsumes all constituent events. XChange^{EQ} supports the thirteen different kinds of temporal relationships specified in Allen's interval temporal logic (Allen 1983).

XChange^{EQ} allows for the definition of two kinds of rules: Deductive and Reactive Rules. Deductive rules cause a new event to be generated based on the composite event statement. Reactive rules cause some other action to be performed within the system. XChange^{EQ} also aims to support declarative semantics. To achieve this goal, XChange^{EQ} avoids employing selection and consumption modes available in other event languages, which cause operators to behave differently based on the context in which they are being used.

XChange^{EQ}'s semantics work directly on streams. This is in contrast with the CQL approach, which utilizes conversion operators to transform streams to temporal relations to perform data manipulation operations, before transforming the relations back to a stream. This roundtrip conversion does not occur in XChange^{EQ}. XChange^{EQ} was also designed with interactions between Web-based systems in mind and thus events are expressed in the XML format. The language Xcerpt (Francois Bry and Schaffert 2003) is used both to specify classes of relevant events and to extract data through variable bindings. The following example is from (François Bry and Eckert 2007), showing a deductive rule based on a conjunction operator with temporal conditions.

```
DETECT earlyResellWithLoss { customer { var C } ,stock { var S } }
ON and {
        event b : buy {{ customer { var C } , stock { var S } , price { var P1 } }} ,
        event s : sell {{ customer { var C } , stock { var S } , price { var P2 } }}
 } where { b before s , timeDiff (b , s )<1hour , var P1>var P2 }
END
```

The composite event detects when a stock is bought and then sold by the same customer at a lower price. The reselling of the stock needs to happen in less than an hour after the original purchase. The buy event and the sell event need to agree on the value of the

variables C and S, which signify the customer and stock, respectively. The conjunction of the two forms a composite event titled earlyResellWithLoss, which includes the customer and stock values. The interval associated with the output event spans the intervals of input events that are composed together. As can be seen in the example, temporal conditions are expressed in the where clause right next to value constraints.

As mentioned earlier, a mechanism is needed to limit the number of events that are being accumulated for negation or aggregation. XChange^{EQ}'s accumulation construct consists of accumulating events while a certain event holds true. The event that specifies the accumulation period can be specified separately as a complex event or can be a relative or absolute timer event. XChange^{EQ} offers several operators that can stretch or shrink the interval of an input event in different directions. The following example from (François Bry and Eckert 2007) displays checking for a negation which is bound by a relative timer event:

```
DETECT buyOrderOverdue { orderId { var I } }
ON and {
        event o : order {{orderId { var I }
        buy {{ }} }} ,
        event t : extend [ o , 1 min ] ,
        while t : not buy {orderId { var I } }
}
END
```

The above query checks when a buy order is overdue. This is defined as an order event *not* being followed by a buy event within 1minute. The event t is a relative timer event. It is created by extending the o event by one minute. It is then used in the while clause to specify the window for the negation. It is important to note that unlike most other languages the while clause is not a separate clause, it is actually specified within the conjunction operator. The while t: not q clause is successful if during the period specified by t, the query specified by q is not satisfied. The period associated with the clause is the same as the t event.

Comparing the XChange^{EQ} model to the CQL model, it could be said that two kinds of windows are supported: an unbounded window, when no limits are placed on the input, and a fixed window when an event interval is specified for accumulation. Sliding windows and all its different variants are not supported. Similarly cumulative-mode like behavior in which arrival of a new event spawns a new window is not supported.

XChange^{EQ} can possibly be augmented by addition of operators that simulate selection mode behaviour: First and Last selection modes can be simulated over a window as aggregate functions without undermining the 'declarative'-ness of the languages.

XChange^{EQ} does not allow direct access to the timestamp values. This is done in the interest of keeping the language simple but can be restrictive on the kind of conditions that can be placed on timestamp values. This issue will be further discussed in Chapter 5.

2.5.7 Tesla

Tesla (Trio-based Event Specification LAnguage) (Cugola and Margara 2010) is a complex event processing language. Tesla supports value and temporal filters, timers, negation, aggregates, and specification of separate and customizable event selection and consumption. Many existing event processing languages only support event selection and consumption in a rigid predefined manner, sometimes combining the two together. The temporal model uses instantaneous timestamps. Whether the timestamps are issued by the event source or the event processing system and the prospect of out of order events are considered a separate issue from the language and are discussed in (Srivastava and Widom 2004).

An example Tesla rule taken from (Cugola and Margara 2010) is:

```
define Fire(Val)
from Smoke() and each Temp(Val > 45)
within 5min from Smoke
where Val = Temp.Value
consuming Temp
```

The define clause defines the name and heading for the output event. The from clause is where the event pattern can be specified. In the above example, the keyword ‘each’, coupled with the ‘within’ condition specifies a selection policy which signifies that each occurrence of Temp that satisfied the value condition in a 5 minute time window will be used for generating a new event. The where clause assigns values to the output parameters. These values may come from the event pattern. The consuming clause is optional and specified the consumption policy. In the above example, the Smoke event is being consumed, which means that no smoke event can be used twice for generating an output. Tesla aims to have precise semantics and is formalized using a temporal first order logic called TRIO (Ghezzi, Mandrioli, and Morzenti 1990).

Tesla’s focus on flexible and precisely defined selection and consumption policies is a welcome one. However, lack of support for interval semantics is a serious drawback. The approach of this research is to treat temporal conditions similar to any regular value conditions. Tesla moves away from this approach by specifying them in separate clauses. The implementation is based on finite state machines which do not offer the flexibility of query plans. Temporal relational operators are not supported as they require support for intervals.

2.5.8 ZStream

ZStream (Mei and Madden 2009) is a more recent event processing language that, like other event processing languages, supports direct specification of temporal patterns using event operators. The queries are evaluated using a query plan-based approach which supports dynamic reordering of the plan to achieve optimized evaluation. The plan is

modeled using a tree structure with leaf nodes corresponding to primitive events and internal nodes representing event operators. Each node has a corresponding buffer, with leaf node buffers corresponding to incoming event streams and internal nodes storing intermediate query results. ZStream is designed so that all buffered events are stored in End-Time order. This facilitates discerning events that are within the time window formed by the final event and the specified length of time. While ZStream has a time window option, it does not offer any of the more complex window behavior found in stream processing languages, such as sliding windows. The language constructs offered in ZStream lack selection and consumption capabilities. ZStream does not support the more complex temporal relationships that are outlined in Allen (Allen 1983).

2.6 Specification and Formalization of Event Languages

Many existing event languages suffer from a lack of proper formalization. This has resulted in differing and ambiguous semantics. However some of the previous work in the event processing area do offer formalizations or attempt to take a formalized look at semantics of other languages. This section takes a brief look at some of the notable efforts in this area.

2.6.1 Zimmer and Unland's Meta-Model

Zimmer and Unland present a generic meta-model (Zimmer and Unland 1999) for describing event languages used in active database management systems. This meta-model is used to model features present in different event languages.

To achieve this model semantics, events are broken into three independent dimensions: event patterns, event selection, and event consumption, which operate on event histories. An event history for any given event type consists of all instances of that event.

The event pattern is concerned with defining the type and order of the events. It also includes the ability to specify a repetition count of a specific event.

Event instance selection determines, in case of the existence of multiple eligible instances of a constituent event, which instance[s] should be selected for generating the output event. The options include the selection of the first or last instances. A third option is using all instances of the input events that do not contradict the event pattern.

Event instance consumption determines whether or not the detection of a composite event, renders events used to detect that event ineligible for further consideration. Two options are possible: *shared* and *exclusive* consumption. Shared consumption means that constituent events may be used for generating further instances of that composite event. In contrast, exclusive consumption indicates that once a constituent event is used to generate an event, it is deleted from the event history of the composite event.

2.6.2 SNOOP and SNOOP-IB

The language SNOOP (Chakravarthy and Mishra 1994) is one of the most widely referenced works in the area of event processing. It also later adopted interval semantics in the form of SNOOP-IB (Adaikkalavan and Chakravarthy 2005). In (Galton and Augusto 2002), the authors analyze the semantics of the original SNOOP in order to make an argument for support of interval-based semantics. They introduce event *occurrence* and event *detection* as two separate concepts. Events may happen over a period of time. This period of time is referred to as the occurrence time of an event. However the system does not become aware of an event until that period is over. The point in time in which the occurrence of an event ends is when it is detected. The notation D(E, t) is introduced to indicate that an event of type E is detected at time t. Likewise, O(E, [t, t′]) indicates that event of type E occurs over the interval [t, t′].

Galton & Augusto continue to define the SNOOP operators using the introduced notations. For instance, the sequence operator is defined as:

$D(E_1;E_2, t) = \exists t' < t\ (D(E_1, t') \wedge D(E_2, t))$

Galton & Augusto illustrate several issues with the detection based semantics of SNOOP; for instance the equivalence of $E_1; (E_2;E_3)$ and $E_2; (E_1;E_3)$ using the above definition for the sequence operator. Consequently interval-based or, as it is called here "occurrence-based", semantics are suggested for SNOOP. Under the occurrence-based semantics, the sequence operator is redefined as:

$O(E_1;E_2, [t_1, t_2]) = \exists t, t'\ (t_1 \leq t < t' \leq t_2 \wedge O(E_1, [t_1, t]) \wedge O(E_2, [t', t_2]))$

Definitions for the rest of the SNOOP operators are also presented in (Galton and Augusto 2002). The ideas in this paper lead to development of SNOOP-IB (Adaikkalavan and Chakravarthy 2005), which incorporates interval-based semantics.

Given the issues with point/detection based semantics, the move towards interval-based semantics is a welcome one. However, incorporating intervals entails dealing with the more complex relationships that are possible between them. The set of operators available in SNOOP is simply not expressive enough to capture these relationships.

2.6.3 SASE

SASE assumes a total order of incoming events over a discrete time model. The formalization of event operators is then done similar to those of SNOOP. In particular, the occurrence of an event is defined as a mapping from the time domain to a Boolean value. The occurrence of an event at a specific time is true if that event occurs at that instant and false otherwise. For example, the ANY operator, which is triggered if any of its contributing events are triggered, is formalized as follows (Gyllstrom et al. 2006):

$\mathrm{ANY}(A_1, A_2, \ldots, A_n)\ (t) \equiv \exists\ 1 \leq i \leq n\ A_i(t)$

which means that the truth value of the mapping for the ANY operator, at time t, is equivalent to the truth value of the occurrence of any one of the specified events at time t. As discussed before, operationally the operators are placed in a pipelined query plan. However, the Sequence and Scan operator is detected using an NFA mechanism. SASE does not support interval-based semantics.

2.6.4 Tesla

The Trio-based Event Specification LAnguage or Tesla (Cugola and Margara 2010) is one of the languages that does provide a formal definition, however this formal definition is based on Temporal Logic. The language and its evaluation were discussed in the previous section. Its formalization is discussed here. TRIO (Ghezzi, Mandrioli, and Morzenti 1990) is a metric, first-order temporal logic, which is used to formally specify Tesla. The meaning of TRIO formulas depend on the current moment of time. TRIO includes two temporal operators called *Futr* and *Past*. The formula Past(A,t) is true if A was true t time units in the past, and Futr is similarly defined. There are also a few other temporal operators which are defined based on Past and Futr. For instance:

$$Alw(A) = A \wedge \forall t\,(t > 0 \rightarrow Futr(A, t)) \wedge \forall t\,(t > 0 \rightarrow Past(A, t))$$
$$WithinP\,(A, t_1, t_2) = \exists x\,(t_1 \leq x \leq t_1 + t_2 \wedge Past(A, x))$$

For example, consider the following Tesla rule from (Cugola and Margara 2010):

```
define Fire(Val)
        from    Smoke() and
                each Temp(Val > 45) within 5min from Smoke
                where Val = Temp.Val
```

This rule raises a Fire event based on Smoke and Temp. For each Temp with value greater than 45 within 5 minutes from smoke, the fire rule is triggered. Below a simplified version of the TRIO formulation for the conjunction operator with the ‘each’ selection policy is presented:

define CE from A and each B within x from A =
Occurs(CE) ↔
(Occurs(A) ∧ WithinP (Occurs(B), Time(A), x))

CE is the composite event, A and B are the input events. Occurs is a predicate that is used to map the truth of a TRIO formula to the detection of an event. Further details of the formalization of Tesla are beyond the scope of this research and can be found in (Cugola and Margara 2010). Tesla does not support interval-based semantics.

2.6.5 XChange^{EQ} and CERA

The work in XChange^{EQ} uses an algebra closely based on relational algebra to describe the operational semantics of the language. This algebra is called Composite Event Relational Algebra (CERA). XChange^{EQ} works with XML data. But for the sake of its operational semantics, it is assumed that the XML data is stored as relational attribute values. The XChange^{EQ} operators are then formalized based on relational operators and concepts.

In CERA, event histories are treated as relations. CERA imposes several restrictions on relational algebra and also introduces several short-hands for ease of writing frequently used constructs.

In CERA, begin and end timestamps of events are treated as attributes in a relation. Projections that result in the omission of the timestamps are not allowed. Direct modification of the timestamp values is also not allowed. As mentioned before, since XChange^{EQ} deals with events that are expressed in XML, CERA adds *Matching* and *Construction* operators that enable mapping of XChange^{EQ} queries to a relational algebra-based model. To simplify discussion of issues, a relational version of XChange^{EQ} called Rel^{EQ} is introduced in (Eckert 2008). In Rel^{EQ} incoming events are assumed to be relational tuples instead of the XML format which is expected in XChange^{EQ}. The following is a simple example of a Rel^{EQ} query from (Eckert 2008).

comp(id, p) ← o : order(id, p, q), s : shipped(id, t), d : delivered(t) o before s, s before d, {o, s, d} within 48

The above event signals a completed order which is defined as an order event followed by a shipped event which itself is followed by a delivered event. Additionally, this sequence of events needs to be completed within 48 hours. The relational algebra expression for the above query can be expressed as:

$$\sigma[\max\{o.e, s.e, d.e\} - \min\{o.s, s.s, d.s\} \leq 48]$$
$$(\sigma[s.e < d.s](\sigma[o.e < s.s](Ro \bowtie Ss) \bowtie Td)))$$

For ease of reading, instead of the traditional subscript notation, brackets are used for specifying the parameters of relational operators. The variables s and e are the beginning and ending timestamps of events, respectively. The above expression assumes full knowledge of every event that has happened in the past or will happen in the future. In reality, we have access to past events but future events are yet to come, and the relational expression needs to be evaluated in a continuous, step-wise manner. CERA and step-wise evaluation of relational algebra expressions will be discussed further in Chapters 5 and 6.

The model presented in (Eckert 2008; François Bry and Eckert 2007) has the following limitations:

- Strict requirements on the timestamps: The output timestamp can only be produced using the merging operator which is required to appear in every rule head which specifies the heading for the output. Since extended projection, which allows for modification of heading attribute values, is not allowed, output timestamps cannot be modified. For example, to modify the starting timestamp or postpone the ending timestamp of a produced event is not allowed.

- No interaction between event data and timestamp data: Since the timestamps can only be accessed in very specific ways, comparison or modification of timestamps based on event data is not possible. This can be a significant shortcoming if the event itself includes data that is temporally relevant.

Further, because specification of temporal conditions is only possible through the use of specific operators, the kinds of conditions that can be expressed on them even without the use of event data is limited. More recent work has tried to ally this problem by providing a longer list of operators, which are more appropriately restricted or open (Walzer, Breddin, and Groch 2008). Such extensions might fine tune the language for a specific application. However, they do not address the root cause of expressivity issues which are a result of hiding timestamps.

2.7 Discussion

An overview of the current status of related work has been presented in previous sections. Work in the stream processing area (A. Arasu, Babu, and Widom 2006; Abadi et al. 2003) aims at filtering of high volume streams of data. A considerable amount of the focus is on the architecture necessary to cope with these high volume requirements. Different approaches have been pursued to handle excess volume when available resources are not sufficient to produce precise results to queries. CQL is the language that forms the basis for much of the work in this area, including commercial products. All of these stream processing languages use the concept of a time-varying relation, which is used to perform the bulk of data manipulation operations in a query. Using time varying relations entails converting an input stream to a relation for processing and then transforming the result back into a stream, which is not very convenient. In stream languages, there is usually little to no support for stream-to-stream operators that can directly detect temporal patterns of events. Additionally, much of the work in stream

processing assigns a unary timestamp to an input stream and thus interval-based semantics are ignored.

Event processing focuses instead on detection of temporal patterns and extraction of order information between different events. Some of the earlier work on processing of event streams is based on the Event portion of an ECA rule in active databases (Chakravarthy and Mishra 1994). Work found in publish-subscribe systems use finite state structures to encode user requested event patterns or subscriptions (Mühl, Fiege, and Pietzuch 2006). The automata structure is well suited for cross query optimization and reducing space requirements for storing and processing of an event query. However, it is inflexible in its order of evaluation of different operators and is unable to formulate alternate evaluation plans. These languages usually do not support value correlation of different events.

SnoopIB expands on Snoop's composite operator-based approach by adding interval semantics to event queries. Lack of interval semantics can lead to ambiguous operator meaning, which has been discussed in (Eckert 2008; White et al. 2007). Snoop also presents the concepts of consumption modes, which are used to reduce the number of active events under consideration at any given time. Closer inspection reveals that consumption modes as discussed in Snoop is in fact a mix of both selection and consumption modes. The use of consumption modes is associated with an entire event definition instead of individual input events which makes its use rather inflexible. Later work in (Sánchez et al. 2003; Cugola and Margara 2010) decouples selection and consumption modes, which leads to clearer semantics. However according to (Eckert 2008) performance benefits of using selection and consumption modes have not been demonstrated yet.

Composite event operators have been the main approach for detecting temporal patterns of events. Composite events operators, however, have consistently been subject to misunderstanding or unclear definitions. Introduction of consumption modes with unclear or confusing semantics has made matters worse. Utilizing interval semantics instead of detection semantics helps clarify some of the inherent issues of this approach, but is still prone to semantic issues as discussed in Section 2.3.

Finally, the work in XChange^{EQ} addresses many of the problems in previous event languages. However, it does not directly support temporal relational operators. Additionally, the values of timestamps are not directly available for querying, which leads to the introduction of a multitude of temporal operators. This makes the language larger and more complex and, at the same time, less flexible.

None of the previous work supports evaluation of temporal relational operators, as presented in (Date, Darwen, and Lorentzos 2002), over streams, and thus the technical challenges associated with evaluation of temporal relational operators over streams of data have not been addressed. Temporal Databases and their associated operators are discussed in Chapter 3.

Chapter 3

TEMPORAL RELATIONAL DATABASE OPERATORS

Temporal Databases (Etzion, Jajodia, and Sripada 1998) are databases that maintain historical data, which is data with an added temporal axis. This temporal axis is modeled through assigning an interval value with each tuple of a relation that is to contain temporal data. Conceptually, the temporal axis associated with the data might reflect an interval in the past, in the future, or currently ongoing. This chapter presents relevant background information on temporal databases with a special focus on the work in (Date, Darwen, and Lorentzos 2002). In future chapters, this background is built upon to present the centric idea of this dissertation, which is incorporation of operators developed in the temporal database realm to an event processing language. Section 3.1 introduces the fundamental concepts of temporal databases. Section 3.2 discussed the TSQL2-based approaches to temporal databases (Snodgrass 1995). Section 3.3 discusses temporal database operators from (Date, Darwen, and Lorentzos 2002) with examples. Section 3.4 provides a basic comparison of the temporal database and event processing settings and discusses the benefits that can be achieved by integrating the two frameworks.

3.1 Temporal Databases

Temporal databases have been a subject of research for decades, but have not yet found their way into standardized bodies nor have they yet gained commercial prevalence. There have been multiple proposals for modeling of temporal databases and associated query constructs.

Two of the most prominent proposals are the approach outlined in (Date, Darwen, and Lorentzos 2002) and the approaches based on TSQL2 (Snodgrass 1995). Both of these proposals support modeling of temporal data through associating an interval with each tuple in a table that is to hold historical data. In general, depending on

the approach, more than a single interval attribute may be used in a relation based on the modeling requirements of the environment. There are, however, two types of intervals, each with a specific purpose, which have been extensively addressed in the literature. These are the *valid time* and *transaction time* intervals.

- Valid time reflects the interval during which the predicate associated with a tuple is assumed to be true. For example, if a tuple reflects the fact that a supplier supplied a part in a city, the associated valid time interval reflects the period during which this was true.
- Transaction time is used to signify the current and historic state of the data in the database. Only data with transaction time values that include the current moment in time are considered to reflect the current state of the database. Conversely, tuples with transaction times in the past reflect the historical state of the database. Transaction time can be used, for instance, to model deletion operations, without actually deleting the underlying data.

3.2 TSQL2-Based Approaches

An approach to temporal databases that has received considerable attention is the TSQL2 approach (Snodgrass 1995) developed by a number of database researchers. A number of different proposals have been offered based on the TSQL2 approach, that adopt the same key concepts. While the TSQL2 approaches are not pursued for the purposes of this research, a brief discussion of TSQL2 is presented here for the sake of completeness.

In the TSQL2 approach, tables that are used for storing temporal data can have one or two temporal attributes. These are used to store the valid time and transaction time of the data as explained above. In the TSQL2 approach, these attributes are hidden from the user and their values cannot be directly accessed. Since both of these attributes are optional, TSQL2 supports 4 different kinds of tables:

- A *bi-temporal table*: This kind of table includes both the valid time and transaction time timestamp attributes
- A *valid-time table*: This kind of table includes the valid time attribute, but not the transaction time attribute.
- A *transaction-time table*: This kind of table only includes the transaction time attribute, but not the valid time attribute
- A *regular table*: This kind of table does not include temporal attributes.

The notable feature of the TSQL2 approach is categorizing operations into three different kinds:

- *Current*: Current operations are only applied to the most recent data.
- *Sequenced*: Sequenced operations are applied to all data, or data at all points of time.
- *Non-Sequenced*: Non-sequenced operations are applied to a specified subset of data.

These different kinds of operations are specified by using what are called *Statement Modifiers*. A simple example from (Darwen and Date 2005) is used below to illustrate this concept. Consider the sample set of data in Figure 3.1.

Table S:

S#	
S1	[d01:d01]
S1	[d05:d06]
S2	[d02:d04]
S2	[d06:d99]
S3	[d05:d99]
S4	[d03:d99]
S6	[d02:d03]
S6	[d06:d09]

Figure 3.1. Sample TSQL2 Table

The first column is the identifier of S, called S#. The second column is a valid time interval in days. The two figures indicate the day the interval begins and ends.

Now consider the following simple query:

```
Select * from S
```

If the current day is d10 and the above query is run in the Current mode, the result will include only tuples that include the current day in their valid time. In this case, the result would be: S2, S3, S4

The same query, if run in the Sequenced mode, would return all distinct S# values. The output would be S1, S2, S3, S4, S6. The result would also include the hidden valid time attribute.

The non-sequenced version of the query would return all S tuples that have ever existed in that table but would not include the hidden valid time attribute.

Compared to the TSQL2, the work in (Date, Darwen, and Lorentzos 2002) is a more natural candidate for this research. The approach adopted in this research requires direct access to timestamp values, which is supported in Date, Darwen and Lorentzos' approach. In contrast, the model presented in (Snodgrass 1995) suggests that the temporal axis be modeled through hidden attributes. Hiding the temporal axis of the data as in (Snodgrass 1995), and only allowing the user to query it through provided operators limits the user's ability to express desirable conditions. For this reason the TSQL2 approach will not be discussed any further in this dissertation.

3.3 Temporal Relational Operators

In this section, temporal relational operators are presented. To define the temporal relational operators, a few generic operators need to be discussed first, which will be presented shortly. The temporal framework and operator definition presented in this section are based on the work in (Date, Darwen, and Lorentzos 2002). When presenting relational formulas, the syntax provided is in the style of Tutorial D (Date and Darwen 2000). Tutorial D is a relational database language used by Date and Darwen in (Date and

Darwen 2000). As the name implies, Tutorial D is developed for educational purposes and the queries are in algebraic format.

In the temporal model at hand, time is assumed to be discrete. This is required for evaluating temporal relational operators which will be demonstrated shortly. Picking the granularity of the time attribute is left as a design decision to the database designer.
In the valid-time model, relations which are to store temporal data, include an interval attribute which consists of a start time and an end time.

3.3.1 The Expand and Collapse Operators

The Expand and Collapse operators are two operators which operate on a set of intervals and return a set of intervals. The resulting set of intervals for each operator are in a particular canonical form which are called the *Expanded* and *Collapsed* forms, respectively.

Two sets of intervals are said to be *equivalent* if the set of all points contained in one is equal to the set of points contained in the other. Consider the following example:

S1 = { [*d01*:*d02*], [*d03*:*d07*], [*d09*:*d10*] }
S2 = { [*d01*:*d01*], [*d04*:*d05*], [*d08*:*d09*], [*d10*:*d10*] }

S1 and S2 are not equivalent as, for instance, S2 contains the point d08 which is absent from S1.

S3 = { [*d01*:*d02*], [*d03*:*d04*], [*d05*:*d07*], [*d09*:*d10*] }

S1 and S3 are equivalent as both contain exactly the following set of points:

{ *d01*, *d02*, *d03*, *d04*, *d05*, *d06*,*d07*, *d09*, *d10* }

The corresponding set of unit intervals is:

S4 = { [*d01*:*d01*], [*d02*:*d02*], [*d03*:*d03*], [*d04*:*d04*], [*d05*:*d05*],
[*d06*:*d06*],[*d07*:*d07*], [*d09*:*d09*], [*d10*:*d10*] }

S4 is equivalent to both S1 and S3 and is said to be in the *expanded form* of S1 and S3.

Formally, if S is a set of intervals, the operator EXPAND, which produces the expanded form of S, is defined as follows: If i and j are intervals, let *i*=[*b*:*e*], be the interval that begins at time *b* and ends at time *e*. Then:

EXPAND(S) ≡ { *i* : *b* = *e* AND (EXISTS *j* ∈ S) (*b* ∈ *j*)}

Since the expanded form of a set of intervals corresponds directly to the points contained in that set, it follows that two set of intervals are equivalent if and only if they have the same expanded form.

While S1, S3 and S4 are all equivalent to each other, they each have a different cardinality. There are numerous other sets of intervals that are equivalent to these three. For instance, consider S5:

S5 = { [*d01*:*d02*], [*d03*:*d04*], [*d03*:*d05*], [*d03*:*d07*], [*d05*:*d05*], [*d06*:*d06*],[*d07*:*d07*], [*d09*:*d10*] }

S5 is also equivalent to S1, S3, and S4 and has a different cardinality from all of them. It should be obvious that there are many different sets of intervals that are equivalent to each other. Of particular interest is an equivalent set of intervals with the minimum possible cardinality. This set is called the *collapsed form* for a set of intervals. For the above example, the collapsed form for S1, S3, S4 and S5 is:

S6 = { [*d01*:*d07*], [*d09*:*d10*] }

The collapsed form of a set of intervals is equivalent to the set of intervals, and consists of no two intervals that overlap or abut each other. For any given set of intervals there is only one, unique, collapsed form.

While the idea of a Collapse operator, which would produce the collapsed form of a set of intervals, is straightforward, formalizing it is relatively complex and requires the introduction of several other operators and thus is not presented here for the sake of brevity. The interested reader is referred to (Date, Darwen, and Lorentzos 2002) for a formal definition of Collapse and a further discussion of these operators.

3.3.2 The Pack and Unpack Operators

Two basic operators used in temporal queries are PACK and UNPACK. These two operators will be presented here. Informally, the UNPACK operator expands each tuple into multiple tuples, such that each of the new tuples has the same value for the non-interval attributes as the corresponding original tuple, and the interval attribute for each new tuple reflects one unit from the original tuple's interval value. The PACK operator takes tuples that are temporally adjacent or overlapping, and reflect the same values in their non interval attribute, and merges them into a single tuple.

The PACK Operator:

Consider the following relation, called Employee_Dept which reflects the amount of time each employee was present in a department:

Employee_Dept

Id	Dept	DURING
e1	CSE	[1001:1003]
e2	CSE	[1002:1003]
e3	ECE	[1006:1008]

In Employee_Dept, Id is the employee identifier, Dept is the department name, and DURING is the duration of time that each specific employee was present in the department.

Now consider the following query:

Return department names and interval pairs, such that the interval represents continuous periods of time during which any employee was present in the department.

The desired result for this query would be:

Dept	DURING
CSE	[1001:1003]
ECE	[1006:1008]

It is clear that the Collapse operator would be useful for answering this query. However collapse can only be directly applied to a set of intervals. A step-by-step derivation of the desired result is demonstrated next using Tutorial D.

A simple projection of Dept and DURING from Employee_Dept yields:

```
WITH (Employee_Dept{ Dept, DURING }) AS T1
```

T1

Dept	DURING
CSE	[1001:1003]
CSE	[1002:1003]
ECE	[1006:1008]

Next, the DURING values associated with each Department are grouped together:

```
WITH (T1 GROUP ( DURING ) AS X) AS T2
```

T2

Dept	X
CSE	DURING [1001:1003] [1002:1003]
ECE	DURING [1006:1008]

The GROUP operator of Tutorial D is similar to the SQL clause GROUP BY, in that it groups attributes together. However, in contrast to GROUP BY, the GROUP operator takes as input the attribute name that is to be grouped. The Collapse operator is then applied:

```
WITH (EXTEND T2 ADD COLLAPSE (X) AS Y) AS T3
```

T3

Dept	X	Y
CSE	DURING [1001:1003] [1002:1003]	DURING [1001:1003]
ECE	DURING [1006:1008]	DURING [1006:1008]

Finally we project the collapsed attribute and ungroup it to produce the desired result:

T3 {ALL BUT X} UNGROUP Y

Dept	DURING
CSE	[1001:1003]
ECE	[1006:1008]

It is desirable to be able to perform the above operations using a single application of a PACK operator. Hence PACK is defined as (Date, Darwen, and Lorentzos 2002):

PACK *R* ON *A* ≡ WITH (*R* GROUP { *A* } AS *X*) AS *R1*,
(EXTEND *R1* ADD COLLAPSE (*X*) AS *Y*)
{ ALL BUT *X* } AS *R2*:
R2 UNGROUP *Y*

Using the PACK operator, the above example query, can now simply be stated as:

PACK Employee_Dept { Dept, DURING } ON DURING

This operation as a whole is known as a *Temporal Projection*.

The UNPACK operator:

Consider that, in addition to the Employee_Dept relation, the following relation called Employee is also present in the database:

Employee

Id	DURING
e1	[1001:1003]
e2	[1002:1004]
e3	[1005:1007]
e4	[1007:1009]

The relation Employee contains employee identifier and interval pairs such that the interval reflects the time during which the corresponding employee was signed in. Now consider the following example query:

> *Return Employee Id and Interval pairs, such that the interval represents continuous periods of time when the corresponding employee was signed in, but was not present in any department.*

Answering this query requires unpacking both the Employee and Employee_Dept relations and performing a difference operation between them. UNPACK is defined very

similarly to PACK, with the exception of using EXPAND instead of COLLAPSE (Date, Darwen, and Lorentzos 2002):

```
UNPACK R ON A ≡  WITH ( R GROUP { A } AS X ) AS R1,
                 ( EXTEND R1 ADD EXPAND ( X ) AS Y )
                                 { ALL BUT X } AS R2:
                 R2 UNGROUP Y
```

Unpacking Employee and Employee_Dept after projecting the Employee Id and DURING, produces the following tables:

Unpacked_Employee

Id	DURING
e1	[1001:1001]
e1	[1002:1002]
e1	[1003:1003]
e2	[1002:1002]
e2	[1003:1003]
e2	[1004:1004]
e3	[1005:1005]
e3	[1006:1006]
e3	[1007:1007]
e4	[1007:1007]
e4	[1008:1008]
e4	[1009:1009]

Unpacked_Employee_Dept

Id	DURING
e1	[1001:1001]
e1	[1002:1002]
e1	[1003:1003]
e2	[1002:1002]
e2	[1003:1003]
e3	[1006:1006]
e3	[1007:1007]
e3	[1008:1008]

Performing a relational difference between the two unpacked relations yields:

```
WITH(Unpacked_Employee MINUS Unpacked_Employee_Dept) AS T1
```

T1

id	DURING
e2	[1004:1004]
e3	[1005:1005]
e4	[1007:1007]
e4	[1008:1008]
e4	[1009:1009]

T1 reflects the desired result in the unit interval format. Packing the result is needed as a final step:

PACK T1 ON DURING

id	DURING
e2	[1004:1004]
e3	[1005:1005]
e4	[1007:1009]

The above order of operations performed to derive the query result is known as a *Temporal Difference*.

PACK and UNPACK can also be performed on several attributes simultaneously. However, for the purposes of this research packing and unpacking on a single interval attribute suffice. For this reason, in some of the discussion, when the relation at hand has a single interval attribute and there is no risk of confusion, the interval parameter will be omitted from PACK and UNPACK.

3.3.3 Generalized Relational Operators

The PACK and UNPACK operators are the basis for defining temporal versions of relational operators. In general, the temporal version of a relational operator is equivalent to performing the relational operation normally on the unpacked operand(s) and then packing the result.

The work in (Date, Darwen, and Lorentzos 2002) argues that the temporal versions of relational operators can be considered as a generalized version of that operator. Thus traditional relational operators can be considered a special case, where no intervals are associated with a tuple.

In the remainder of this section the formulas are provided from (Date, Darwen, and Lorentzos 2002) along with examples to demonstrate the use of different temporal operators:

Difference

An example query for temporal difference was demonstrated in the previous section. Temporal difference is formalized as:

```
USING ( ACL ) ◄ R1 MINUS R2 ► ≡
                PACK ( ( UNPACK R1 ON ( ACL ) )
                          MINUS
                          ( UNPACK R2 ON ( ACL ) ) )
                ON ( ACL )
```

where R1 and R2 are union-compatible relations (have the same heading), and ACL is a list of interval type attributes. A single interval attribute suffices for the purposes of this dissertation. Date et al. consider temporal relational operators as extensions of the standard relational operators. The USING clause is used as a modifier which specifies the behavior of the operator by specifying an interval attribute. The ◄*expression*► is used to delimit the expression which is affected by the USING clause.

Union

Union is defined as:

```
USING ( ACL ) ◄ R1 UNION R2 ► ≡
                PACK ( ( UNPACK R1 ON ( ACL ) )
                          UNION
                          ( UNPACK R2 ON ( ACL ) ) )
                ON ( ACL )
```

where R1, R2, and ACL are the same as in the difference formula. Performing the unpack step does not affect the result of a temporal union, thus we could simplify the above formula as:

```
PACK
( R1 UNION R2 )
ON ( ACL )
```

Given the following relations:

R1

DURING
[1002:1003]
[1004:1004]
[1007:1007]
[1009:1009]

R2

DURING
[1005:1006]

A temporal union produces the following result:

USING (DURING) ◄ R1 UNION R2 ►

DURING
[1002:1007]
[1009:1009]

Intersect

Temporal intersect is defined as:

```
USING ( ACL ) ◄ R1 INTERSECT R2 ► ≡
                PACK ( ( UNPACK R1 ON ( ACL ) )
                          INTERSECT
                          ( UNPACK R2 ON ( ACL ) ) )
                ON ( ACL )
```

where R1, R2 and ACL are the same as in union and difference. Consider the following relations:

R1

DURING
[1001:1001]
[1002:1004]
[1007:1009]

R2

DURING
[1003:1008]

A temporal intersect of R1 and R2 yields the following result:

USING (DURING) ◄ R1 INTERSECT R2 ►

DURING
[1003:1004]
[1007:1008]

Project

The temporal version of project, is defined as:

```
USING ( ACL ) ◄ R { BCL } ► ≡
    PACK ( ( UNPACK R ON ( ACL ) ) { BCL } )
    ON ( ACL )
```

where R is the input relation, ACL must contain only interval type attributes, and BCL is the list of attributes that need to be projected, where ACL $\subseteq$ BCL.

An example of a temporal project query was already presented in the discussion of the PACK operator. In practice, the unpacking step is not necessary when performing a temporal project, thus temporal project can be defined more succinctly as:

USING (*ACL*) ◄ *R* { *BCL* } ► ≡
PACK (*R* { *BCL* })
ON (*ACL*)

Restrict

The temporal version of restrict, which is also known as select, is defined as:

USING (*ACL*) ◄ *R* WHERE *p* ► ≡
PACK ((UNPACK *R* ON (*ACL*)) WHERE *p*)
ON (*ACL*)

where R is the input relation and *p* is the restrict condition, and ACL must contain only interval attributes. The following example illustrates the use of the temporal restrict operator:

R

id	DURING
e2	[1004:1006]
e3	[1005:1005]
e4	[1007:1009]

USING (*ACL*) ◄ *R* WHERE
DURING = ([1005 : 1005])
OR DURING = ([1008 : 1008]) ►

id	DURING
e2	[1005:1005]
e3	[1005:1005]
e4	[1008:1008]

In contrast performing a regular restrict instead of the temporal version would have returned the following result:

id	DURING
e3	[1005:1005]

Join

Temporal join is defined as:

```
USING ( ACL ) ◄ R1 JOIN R2 ► ≡
    PACK
            ( ( UNPACK R1 ON ( ACL ) )
            JOIN
            ( ( UNPACK R2 ON ( ACL ) )
    ON ( ACL )
```

Any attributes specified in ACL should be of interval type and also be present in both R1 and R2. Since the join operator employed in the definition is the natural join, all attributes specified in ACL will be the basis for the join, along with any other shared attributes between R1 and R2. If R1 and R2 have the same exact heading, performing a join operator on them would be equivalent to performing an intersection operator.

In general, the approach presented in Date allows for more than a single interval attribute to be present in a relation, but for the sake of simplicity in the above discussion a single interval attribute has been assumed. The Employee and Employee_Dept example from before is reproduced below for reference:

Employee

id	DURING
e1	[1001:1003]
e2	[1002:1004]
e3	[1005:1007]
e4	[1007:1009]

Employee_Dept

id	Dept	DURING
e1	CSE	[1001:1003]
e2	CSE	[1002:1003]
e3	ECE	[1006:1008]

Consider the following query:

> *Return Employee id, Department name and intervals such that the interval reflects the continuous periods of time during which the employee was signed in and present in the corresponding department.*

The result of this query can be calculated by a simple application of the temporal join operator. Unpacking the two relations yields:

Unpacked_Employee

id	DURING
e1	[1001:1001]
e1	[1002:1002]
e1	[1003:1003]
e2	[1002:1002]
e2	[1003:1003]
e2	[1004:1004]
e3	[1005:1005]
e3	[1006:1006]
e3	[1007:1007]
e4	[1007:1007]
e4	[1008:1008]
e4	[1009:1009]

Unpacked_Employee_Dept

id	Dept	DURING
e1	CSE	[1001:1001]
e1	CSE	[1002:1002]
e1	CSE	[1003:1003]
e2	CSE	[1002:1002]
e2	CSE	[1003:1003]
e3	ECE	[1006:1006]
e3	ECE	[1007:1007]
e3	ECE	[1008:1008]

Joining the two unpacked relations yields the following result:

Unpacked_Employee JOIN Unpacked_Employee_Dept

id	Dept	DURING
e1	CSE	[1001:1001]
e1	CSE	[1002:1002]
e1	CSE	[1003:1003]
e2	CSE	[1002:1002]
e2	CSE	[1003:1003]
e3	ECE	[1006:1006]
e3	ECE	[1007:1007]

Packing on DURING produces the desired result:

id	Dept	DURING
e1	CSE	[1001:1003]
e2	CSE	[1002:1003]
e3	ECE	[1006:1007]

It should be clear at this point that temporal relational operators give us a powerful tool to query temporal data. Temporal relational operators support the expression of queries that would otherwise be hard or impossible to express. The aim of this dissertation is to bear this expressive power on the temporal aspect of event data, which has often been neglected in previous work.

3.4 Temporal Operators in an Event Environment

Temporal Databases specifically address the issue of querying temporal data. As demonstrated in (Date, Darwen, and Lorentzos 2002), queries involving temporal data, such as those presented in the previous section, are not easy to express using standard relational algebra operators.

The goal of TEQL is to enable application of temporal operators as developed in the database context in the event environment based on the XChange^{EQ} relational framework. The direction taken is to try to approach the more open nature of temporal databases in the interest of achieving more expressivity, while preserving the qualities of the XChange^{EQ} framework, which enable evaluation of relational statements over event streams.

TEQL also specifically adds the temporal relational operators presented in Section 3.2 to enable new kinds of queries in the event environment.

In this section, we motivate the choice of temporal database queries as an expressive means for querying events by way of comparison to some of the important previous approaches. These approaches are presented in the following categories:

- CQL: CQL is one of the most important languages in the area of stream processing languages and forms the basis for many other languages developed in this area.We use CQL (A. Arasu, Babu, and Widom 2006) as a representative of stream processing languages.
- Traditional event processing languages: Most of these languages have used a composition-operator based approach. The issues with this approach were discussed in detail Section 2.3.

- XChange^{EQ}: While XChange^{EQ} falls in the category of event processing languages, it differs in many of the features that it provides from traditional event languages.

Since the work in this dissertation is based on the XChange^{EQ} approach, the ways in which the integrated approach presented in this dissertation improves upon XChange^{EQ} are emphasized.

Table 3.1 summarizes the characteristics of CQL, traditional event processing languages, XChange^{EQ}, and Temporal Databases.

Features/Language Categories	CQL	Traditional Event Languages	XChange^{EQ}	Temporal Databases
Unambiguous formal semantics	✓		✓	✓
Interval-based data		Some	✓	✓
Support for Allen's operators		Some	✓	✓
Relational query model	✓		✓	✓
Separation of temporal conditions from output composition			✓	✓
Direct access to timestamps	Possible through extension			✓
Support for temporal relational operators				✓
Support for multiple attributes with temporal data				✓
Control over output timestamp value				✓

Table 3.1 Language Features in CQL, Event Processing, XChange^{EQ} and Temporal Databases

These characteristics are discussed below.

Unambiguous Formal Semantics

Many of the traditional event languages are not properly formalized or suffer from ambiguous and/or inconsistent semantics. This leads to confusion in use of the language and unintended behavior. These issues have been addressed in Section 2.3. Conversely proper specification of a language leads to clear semantics and predictable behavior. XChange^{EQ} is formalized using a tailored version of relational algebra called the Composite Event Relational Algebra (CERA). The temporal database operators presented in Section 3.2 are also formalized based on relational algebra and can be considered a super-set of relational algebra. TEQL integrates temporal database features with the CERA algebra. Extensions to CERA will be presented in Chapter 5 and the specification of TEQL operators will be presented in Chapter 6.

Interval-Based Data

Traditionally, most of the languages in the areas of event and stream processing have failed to support interval-based semantics. Interval-based events have only recently been addressed in event languages such as XChange^{EQ}. Temporal databases support both a point-based and an interval-based specification of timestamp values. Many applications require support for interval-based data. Interval-based semantics also allows for the capturing of more complex relationships among temporal data items.

Support for Allen's Operators

XChange^{EQ} is one of the first event languages to support the specification of Allen's relationships between intervals, which is also supported in temporal databases. These operators were not supported in traditional event and stream processing languages due to their lack of support for interval-based data. Allen's operators encompass all of the different relationships that are possible between two specific intervals.

Relational Query Model

Many of the traditional event languages adopt a composition-operator-based approach for specification of the event query. In contrast, stream processing languages based on CQL and the XChange^{EQ} event processing language utilize a select-from-where query structure, which is the familiar format found in database and temporal database queries.

Separation of Temporal Conditions from Output Composition

Due to their adoption of composition operators, traditional event languages combine the specification of temporal conditions with the event specification. This approach fails to adequately address the temporal aspect of events. Older point-based event processing systems allowed simple ordering of events through operators such as sequence and neglect to treat the timestamp of events as part of the data.

Direct Access to Timestamps

With the emergence of interval-based systems such as XChange^{EQ}, there has still been a tendency to hide the timestamp values and to limit the specification of temporal conditions through the use of a specific list of predefined operators. Due to the complexity of relationships possible between intervals, this results in constriction of expressivity in these languages. In contrast, with databases in general, all data, including meta-data, is visible and can be queried.

Support for Temporal Relational Operators

Aside from allowing direct access to timestamp values for specification of conditions, temporal databases specifically address the issue of querying temporal data through the introduction of temporal versions of relational operators. These operators and the queries that they enable were discussed at length in Section 3.2 and have not been previously addressed in an event processing environment.

Support for Multiple Attributes with Temporal Data

In the XChange^{EQ} model, only a single timestamp value can be present in any tuple. Since this timestamp value is hidden, it is not possible for any temporally relevant value present in the remainder of the event heading to interact with the timestamp value by way of comparison or modification. In contrast, the database model, and by extension the temporal database model, does not discriminate about the kind of data that can be present in a tuple and how the different attributes in a tuple interact with each other in a query specification. Multiple interval attributes are permitted.

Control Over Output Timestamp Value

In XChange^{EQ}, the output timestamp value is generated internally and cannot be directly affected through the query specification. In the temporal database model, the user is free to adjust values of data stored in relations. Likewise in a temporal database query, output values can be completely controlled. For an event language, absolute control over the output timestamp is not possible. However, as will be discussed in Section 5.3, adjusting the output timestamp can be useful as a means to control the time at which an event is generated.

In summary, temporal relational database operators provide a powerful tool for querying temporal data. These operators enable new kinds of queries that are difficult to express using traditional database operators and have not been previously addressed in an event setting. Moreover, they have been formalized based on the well understood relational model. A comparison of temporal database operators to some of the more relevant efforts in the areas of event and stream processing was presented to motivate the integrated approach for querying events. The following chapter presents the TEQL language, which incorporates these powerful operators for querying events.

Chapter 4

LANGUAGE DESIGN FOR TEMPORAL EVENT QUERIES

As seen in the related work section, there have been many different proposals for event and stream processing languages. Many of these proposed approaches suffer from ambiguity or lack of expressiveness. While advances have been made with regards to formal specification of languages, a new focus on interval-based events brings the expressivity issues to the forefront. The approach of this research is to build upon the strong theoretical foundation of relational and temporal relational databases to avoid the problems of ambiguity found in many other event specification languages. Furthermore TEQL enables the use of temporal relational operators in an event setting which leads to new expressive capabilities not available in previous event languages. Enabling temporal relational operators specifically targets querying of interval-based data, an issue that has only recently been targeted in event processing languages. In this chapter, the TEQL language is presented.

The following subsections present the different language features and discuss issues related to these features. Section 4.1 presents an overview the TEQL features. Section 4.2 explains the characteristics of events in TEQL. Section 4.3 presents the basic TEQL query structure. Section 4.4 presents the MERGE function which calculates the output timestamp value. Section 4.5 presents the interval operators and other temporal conditions possible in TEQL. Section 4.6 presents windowed sources and why they are needed. Section 4.7 presents the syntax for temporal relational operators and explains how they are applied in windowed sources. Section 4.8 provides a summary of the chapter.

4.1 Overview of TEQL Features

The TEQL language design builds upon at least three separate bodies of work. The overall structure and feel of the language follows the SQL language structure. The event processing model is based upon the relational version of XChange^{EQ} called Rel^{EQ} (Eckert 2008). However, access to input and output timestamps is also provided for stronger expressivity. Finally temporal relational operator specifications are based on the work of Date, Darwen and Lorentzos (Date, Darwen, and Lorentzos 2002).

The basic query structure of TEQL follows the select-from-where clause found in SQL based languages. Events are treated as relations in the query language. However, the queries themselves are evaluated incrementally on streams of events. Incremental evaluation of TEQL is discussed in Chapter 6. TEQL supports specification of interval relationships as defined in (Allen 1983). Furthermore, following Codd's *information principle*, TEQL treats all data as relational data. As a result, TEQL does not discriminate between temporal and non-temporal data and, instead, supports extra operators for expressing queries over temporal data. Direct access to timestamp values is supported, which is useful for expressing conditions between two or more points in time. In contrast, Allen's operators only support comparisons between pairs of intervals.

TEQL enables non-monotonic and blocking operators through the use of an expressive window operator. Furthermore, TEQL allows controlled manipulation of the output timestamp through the use of the MERGE function.

TEQL is an effort to design a language that follows relational principles not only internally, but in the expressivity of the language as well. Besides this, perhaps the most important contribution of TEQL is enabling the use of temporal relational operators in an event setting. The temporal relational operators supported in TEQL are: PROJECT, RESTRICT, UNION, INTERSECT, DIFFERENCE and JOIN. There operators enable

temporal queries that are difficult to express using standard relational algebra, and have previously not been supported in an event environment.

The next section discusses the characteristics of events in TEQL.

4.2 Events in TEQL

In TEQL, every event belongs to a predefined type. Similar to a database relation, each event type is defined through a heading, which consists of a set of attribute name and data type pairs.

Incoming events are stored in a *virtual event history relation* pertaining to that specific event type.

Physically, the event history relation is divided into two separate relations: 1) The delta, which pertains to events that have just arrived, and 2) the actual event history, which is a store for events that have arrived at an earlier point in time. As far as the TEQL query language is concerned, this division is not visible and in TEQL all queries are expressed over the virtual event history relation. Thus for the purposes of this chapter, we will drop the 'virtual' designation and simply refer to the event store as the event history relation, or simply as the event history. The physical division is a matter of query evaluation and will be discussed in Chapter 6.

Individual events are equivalent to tuples in relations. Arrival of each event is marked by its insertion into the appropriate event history.

The heading of an event is represented as:

$\{a_1, a_2, \ldots, a_n, \text{DURING}\}$

where $a_1, a_2, \ldots, a_n$ are the name of the event attributes, each having a specific type. DURING, is also an attribute name that must be present in every event and represents the timestamp of the event. The data type of DURING is the *interval type*.

The interval type, is a structured type which consists of a begin time, *b*, and an end time, *e*. The values of *b* and *e* are points in time. The actual data types of *b* and *e* might be such types as milliseconds, hours or days. The granularity of time chosen depends on the application scenario and is flexible from one application to another. For each application however, a discrete granularity of time needs to be assumed. In order for [b : e] to represent a valid interval of time, it is necessary that b ≤ e.

The events themselves are of two different kinds:

- Primitive: These events are provided by external sources such as sensor readings and serve as an input to the system. Primitive events are assumed to be instantaneous. Thus for primitive events, the begin and end time of DURING have the same value: b=e.
- Complex: Complex events are generated as a result of running queries over primitive and/or other complex events. The value of the timestamp for each generated complex event depends on the events from which it has been derived. For complex events b is always less than or equal to e: b ≤ e .

4.3 TEQL Query Structure

The TEQL event query structure is based on the SQL select-from-where format:

DEFINE *query_name* (*typed_attribute_list*) AS
SELECT [*attribute_list*], MERGE([*b_offset*], [*e_offset*]) AS DURING
FROM *event_list*, [WINDOWED_SOURCE]
[WHERE *condition_list*]

A TEQL query definition consists of four clauses:

- DEFINE is used to specify a name for the query (*query_name*). The heading of the query is also defined here through specifying a *typed_attribute_list*. This list consists of attribute name and type pairs. The attributes must appear in the

SELECT clause. The DURING attribute is automatically included in the heading and is used to store the timestamp of the event.

- SELECT is used to specify the list of attributes (*attribute_list*) to be included in the query result. The *attribute_list* consists of the list of attributes a_1, a_2, ... , a_n, that are projected from the attributes of the events specified in the FROM clause. The SELECT clause should always include an attribute called DURING, which signifies the interval associated with the output of the query. The timestamp for the output needs to be specified using the interval construction function called MERGE(). The MERGE function will be discussed in Section 4.4
- FROM is used to specify the different event streams that serve as data sources for the query. The *event_list* is the list of input event types. Additionally, windowed, non-monotonic sources can be specified in the FROM clause. Windowed sources will be discussed in Section 4.6.
- WHERE is used to specify the *condition_list* that includes any value conditions that can be specified over the attributes of the events in the *event_list*. Interval operators based on relationships as specified in Allen (Allen 1983), such as before or after, can be specified over the event types included in the event_list. Interval operators and other temporal conditions will be discussed in Section 4.5.

4.4 The MERGE function

By default, the merge function calculates the output interval-based on the semantics of the *merge operator* as defined in XChange^{EQ}. Informally, the merge operator takes the intervals of all input events that are used to generate an output event and assigns the interval that spans the smallest start time and the greatest end time as the output interval.

TEQL aims to give the user more direct control over the output timestamp. This is achieved through using the operands of the MERGE() function. Similar to the window

specification, the user can specify offsets on the values of the output duration. These offsets are applied to the default output of the merge operation.

MERGE(*b_offset*, *e_offset*)

The b_offset can be any integer value, but the e_offset can only be non-negative values. The reason for this is that current results should always be calculated using current events. If the end time of an event is allowed to be reduced, then an event currently being generated actually belonged in the past, and could have affected the output of other events that reference this event type. As such, shifting the occurrence time of an event at the current instant of time to a value less than the current instant of time should not be allowed. The formal definition of MERGE will be presented in Section 5.5.

Throughout this chapter, the example scenario used to demonstrate TEQL is based on the linear road example environment from (Arvind Arasu et al. 2004), which is a standard scenario for stream processing. The linear road scenario has been slightly modified for the sake of demonstrating event behavior. In our modified version of the linear road example, there is a single input event stream called PosSpeedStr2 that contains notifications sent by a vehicle while it is being driven through different highway segments:

PosSpeedStr2(vehicleId, speed, segment, dir, hwy, [b:e])

The vehicleId is the identifier for the vehicle, speed reflects the vehicle speed, segment indicates the highway segment number, dir is the direction of the vehicle's movement, and hwy is the highway number. Each highway is divided into one mile segments. Events are only generated when a vehicle enters a segment and reports the segment number.

Since we are operating in an interval-based setting, an interval attribute is used for the timestamp. The attribute named DURING of type interval, holds the primary timestamp value for any event. The interval type attribute has two components, b and e,

which reflect the begin and end time of that interval. Because PosSpeedStr2 reflects an instantaneous reading, the length assigned to the interval is zero (i.e. b=e).

Consider the following example query:

Query 1: Return PosSpeedStr2 readings that reflect a vehicle with a speed greater than 75 mph.

This query can be expressed in TEQL through a simple value condition in the WHERE clause:

```
DEFINE speedingStr (vehicleId, speed, segment, dir, hwy, DURING) AS
SELECT
    P1.vehicleId, P1.speed, P1.segment, P1.dir, P1.hwy, MERGE() as DURING
FROM
    PosSpeedStr2 P1
WHERE
    P1.Speed > 75
```

For each event arriving in PosSpeedStr2, if the value of Speed is greater than 75, a new speedingStr event is generated:

PosSpeedStr2

vehicleId	Speed	Segment	Dir	Hwy	DURING
100	70	7	1	12	[100:100]
100	76	8	1	12	[105:105]
200	80	9	1	12	[106:106]
200	80	10	1	12	[110:110]

SpeedingStr

vehicleId	Speed	Segment	Dir	Hwy	DURING
100	76	8	1	12	[105:105]
200	80	9	1	12	[106:106]
200	80	10	1	12	[110:110]

In speedingStr, MERGE is called with no parameters. This means that b_offset and e_offset both assume the default value of zero. Since only one input event instance is relevant to the generation of each output event, the maximum end time stamp and minimum begin timestamp for each output event are the same as the original values and the result of MERGE is the same as the input event interval. To delay the end time of the Speeding event by 5 units of time, the DURING attribute could be specified as MERGE(0,5). Providing a positive value for e_offset results in a delayed generation of the query result:

SpeedingStr

vehicleId	Speed	Segment	Dir	Hwy	DURING
100	76	8	1	12	[105:110]
200	80	9	1	12	[106:111]
200	80	10	1	12	[110:115]

The behavior of MERGE in more complex situations will be explained as new examples are presented in the following sections.

In cases when the underlying query only has a single interval attribute, specifying the MERGE function may not seem necessary if the user does not wish to modify the output timestamp because applying the merge behavior to a single pair of timestamps does not have any effect. However, inclusion of the MERGE function along with the DURING attribute in the select list is used for consistency.

4.5 Interval Operators and Temporal Conditions

TEQL supports the specification of interval operators and other temporal conditions in the WHERE clause. Allen (Allen 1983) formalizes the different relationships that are possible between two intervals.

The different kinds of temporal relationships specifiable in TEQL are similar to XChange^{EQ} and are described below:

- i before j: event i ends before event j starts
- i contains j: event i contains event j
- i overlaps j: event i starts before event j but ends within j
- i meets j: the end time of i is the same as the begin time of j
- i starts j: i and j begin simultaneously but i ends before j does
- i finishes j: i and j end at the same time and i's begin time is after j
- i equals j: the intervals for i and j are the same

The first 6 relationship above also have a reverse version: after, contained by, overlapped by, met by, started by, finished by. This brings the total number of

relationships possible between two intervals to 13. For the sake of brevity only the above 7 operators will be discussed and included in the language. The reverse versions can be specified by simply reversing the order of operands in the operator.

Most event languages do not expose timestamp values for direct querying. In some interval-based event languages, operators based on Allen's relationships can be used to specify temporal conditions between different events, for example A before B.

This approach works well for specifying relationships between two intervals. However if we consider intervals as data values, other relationships become possible which are not easily captured using Allen's operators. For instance, consider the case when the begin time of event A needs to be greater than or equal to the begin time of event B. Relationally, it is simple to express this condition as: b.b ≤ a.b However, if we are restricted to using Allen's operators only, the expression becomes:

```
(a after b) or (a during b) or (a overlaps b) or (b meets a) or (b starts a) or (a starts b) or
(b finishes a) or (a finishes b) or (a equals b)
```

Besides being extremely tedious to formulate, the above expression is also readily prone to errors. This is why TEQL supports specification of temporal conditions directly over timestamp values.

Similar to SQL, predicates over event attributes can be specified in the WHERE clause and can be combined using AND and OR. The timestamp interval attribute DURING is also a mandatory event attribute and can participate in these conditions. The begin time of event alias e1 being greater than the begin time of event alias e2 can be expressed as:

```
e1.DURING.b <= e2.DURING.b
```

TEQL also supports specification of Allen's interval operators: BEFORE, CONTAINS, OVERLAPS, MEETS, STARTS, FINISHES and EQUALS. These operators need to be specified over two intervals.

The following query demonstrates the use of the BEFORE operator:

Query 2: Return interval-based events reflecting the amount of time that a vehicle spent in each segment:

```
DEFINE segStr (vehicleId, segment, DURING) AS
SELECT
    P1.vehicleId, P1.segment, P1.dir, P1.hwy, MERGE() as DURING
FROM
    PosSpeedStr2 P1, PosSpeedStr2 P2
WHERE
    P1.DURING BEFORE P2.DURING,
    P1.vehicleId = P2.vehichleId,
    P1.segment+1 = P2.segment
```

The above query performs a self-join of PosSpeedStr2 over the vehicleId number, with P1 and P2 serving as the two different aliases of PosSpeedStr2. P1 needs to occur before P2 and P2 needs to belong to the segment immediately after P1. The output interval-based event is called segStr and retains the values of vehicleId, direction, and highway number from P1. The output event stream uses the entry segment number to tag the entire segment. The tables below demonstrate the events generated by the segStr running against a sample stream of PosSpeedStr2 events:

PosSpeedStr2

vehicleId	Speed	Segment	Dir	Hwy	DURING
100	70	7	1	12	[100:100]
100	76	8	1	12	[105:105]
200	80	9	1	12	[106:106]
200	80	10	1	12	[110:110]

segStr

vehicleId	Segment	DURING
100	7	[100:105]
200	9	[106:110]

The output timestamp duration is calculated using the mandatory MERGE function. Calculation of MERGE is illustrated below for two events in PosSpeedStr2 that satisfy the WHERE clause conditions:

vehicleId	Speed	Segment	Dir	Hwy	DURING
100	70	7	1	12	[100:100]
100	76	8	1	12	[105:105]

Call the first tuple t_1 and the second tuple t_2. It is clear that:

t_1.vehicleId=t_2.vehicleId and t_1.segment+1 = t_2.segment

Additionally,

t_1.DURING BEFORE t_2.DURING

is true, because (t_1.DURING.e=100) < (t_2.DURING.b=105). Hence, all conditions in the WHERE clause are true for this pair of tuples. The output DURING interval is generated by the call to the MERGE function as follows:

- For the begin time, the minimum of the start times for events involved in generating the output must be considered. The minimum of { t_1.DURING.b=100 , t_2.DURING.b=105 } is 100.
- For the end time, the maximum of the end times for events involved in generating the output is considered. The maximum of { t_1.DURING.e=100 , t_2.DURING.e=105 } is 105.

Since the offset values are both zero the output DURING value is [100 : 105]

Since we have a condition in the WHERE clause stating that P1.DURING must be BEFORE P2.DURING, MERGE will always take its begin timestamp from P1 and its end timestamp from P2. Thus in the stated query, the value of DURING in the output event reflects the period during which a vehicle was present in the segment.

4.6 Windowed Source

Relational operations include non-monotonic or blocking operators that cannot be directly supported in an event environment. In TEQL, non-monotonic operators are supported through the use of windows that place limits on the input to these operators and force them to produce an output at the end of the window. Temporal relational operators are also non-monotonic and thus need to be specified within a window. Non-monotonicity of temporal database operators will be discussed in Section 5.4.

TEQL windows are based on the duration intervals associated with events. The logic behind this is that desirable intervals can be constructed using the query language, and thus an event interval itself is a flexible way for specification of intervals. In XChange^{EQ}, several different operators are introduced in order to allow the manipulation of the source event interval in different directions. A more flexible and succinct operator is introduced for this purpose in TEQL. This operator is called the WINDOWED_SOURCE operator. While it is used to specify bounds on other operations, it simultaneously acts as an event source to the rest of the query.

There are four different data sources that are involved in or interact with a window specification:

1. The parent query, which includes other event sources besides the window

2. The interval source, which supplies the window boundaries

4. The result of the underlying operation which is bound by the window

4. The window result, which serves as a data source to the parent query

The WINDOWED_SOURCE is specified as:

WINDOWED_SOURCE((interval_source_alias, [b_offset, e_offset]), {window_attributes}, [*temporal_operation* | *aggregate_operation*])

The operands of the WINDOWED_SOURCE operator are described below:

- Interval_source_alias: The first operand is an input event stream, which forms the basis for the window interval.
- [b_offset, o_offset]: Specification of this pair of offsets is optional. The pair can be used to apply offsets to the begin and end timestamps of the input event stream.
- {window_attributes}: This operand specifies the list of attributes that need to be projected from the window statement. These form the heading for the window output. In addition to these attributes, each window also has a mandatory

DURING attribute. The value of DURING for a window is calculated by applying offsets to the timestamp of the interval source event stream.

- The fourth and final operand is the statement that is temporally bound by the window. This can be a temporal relational operator or an aggregate operation.

The output of the WINDOWED_SOURCE operator is a data source that participates in the parent query like any other event source.

- The data in the WINDOWED_SOURCE operator is the result of applying the underlying temporal operator bound by the window boundaries
- The heading of WINDOWED_SOURCE consists of the attributes specified in window_attributes plus a DURING attribute. The DURING attribute for a window_source is the result of applying [b_offset, e_offset] to interval_source_alias.DURING.

Temporal operations will be discussed in Section 4.7. The aggregate operation can be specified using one of the following 5 functions:

- Average, specified using AVG()
- Count, specified using COUNT()
- Minimum, specified using MIN()
- Maximum, specified using MAX()
- Sum, specified using SUM()

Aggregate operations are specified using the following syntax:

[*grouping_attributes,*] *aggregate_function*() AS *aggregate_function_alias, event_source*

where *aggregate_function*() is one of the above 5 functions. *aggregate_function_alias* is the attribute name assigned to the value returned by the aggregate function. Specifying *grouping_attributes* is optional. If specified, the grouping attributes will be used to group attributes with the same value together. If no grouping attribute is specified, the grouping

occurs over the window. Finally *event_source* specifies the name of the event source over which the aggregation operation occurs.

The following example query illustrates the use of the WINDOWED_SOURCE operator in conjunction with an aggregate function:

Query 3: Every 90 seconds, return vehicleID, speed pairs reflecting the top speed for each vehicle.

```
DEFINE maxSpeed (vehicleId, topSpeed, DURING) AS
SELECT vehicleId, topSpeed, merge() as DURING
FROM
        90_second_timer nst,
        WINDOWED_SOURCE (nst, [0:0], {*} ,
                vehicleId, MAX(speed) AS topSpeed, PosSpeedStr2)
```

This example uses two input event streams. The PosSpeedStr2 stream and a timer event called the 90_second_timer. The 90_second_timer event is a predefined internal event which occurs every 90 seconds. The first few timer events in 90_second_timer are illustrated below:

90_second_timer

DURING
[001:090]
[091:180]
[181:270]
[271:360]
[361:450]
[451:540]

Considering the same input for PosSpeedStr2 as before:

PosSpeedStr2

vehicleId	Speed	Segment	Dir	Hwy	DURING
100	70	7	1	12	[100:100]
100	76	8	1	12	[105:105]
200	80	9	1	12	[106:106]
200	80	10	1	12	[110:110]

The result of the query is illustrated below:

maxSpeed

vehicleId	Speed	DURING
100	76	[091:180]
200	80	[091:180]

Since the window boundaries encapsulate the timestamp of PosSpeedStr2, MERGE returns the window boundaries as the output timestamp.

4.7 Temporal Relational Operators

The temporal operators incorporated into TEQL are temporal JOIN, RESTRICT (SELECT), PROJECT, UNION, INTERSECT and DIFFERENCE. This section presents the syntax for these operators. The input relation[s] to these operators must have an interval typed attribute called DURING. All of these operators need to be specified within a window.

Applying windows to temporal operators entails renaming the DURING attribute of the input relations. This is necessary to uphold the temporal preservation property that enables incremental evaluation of the temporal operators. The concept of temporal preservation and its application to TEQL is discussed in more detail in Chapter 5. Every tuple in each operand is tagged with the window interval associated with it. Since the window interval is already named DURING, the operand intervals are renamed to TEMP_DURING to avoid a conflict.

Figure 4.1 illustrates the steps that are performed when evaluating temporal operations within a WINDOWED_SOURCE specification. For each operand, the DURING attribute is renamed to TEMP_DURING. Then each tuple is tagged with the interval value of the window, called DURING. Finally, the temporal operation is performed by UNPACKING TEMP_DURING, performing the operation, and re-packing TEMP_DURING. The result of a WINDOWED_SOURCE will include two interval attributes: TEMP_DURING which reflects the temporal data gained from performing the

temporal operation, and DURING which is the window over which the operation was performed.

The temporal relational operators are discussed and illustrated through the use of example queries in the following subsections.

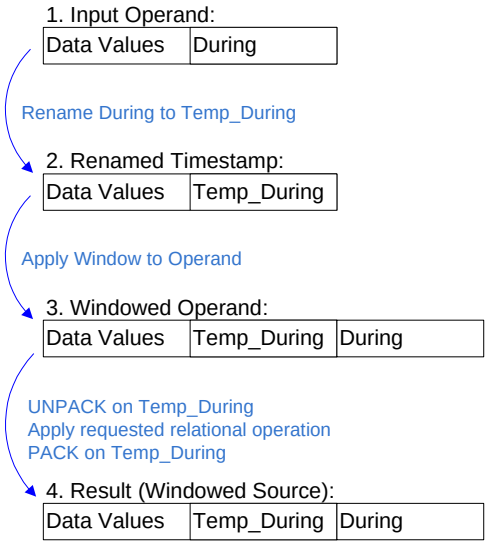

Figure 4.1. Performing a Temporal Operation within a WINDOWED_SOURCE

4.7.1 Temporal Projection

A temporal projection takes as input a relation and a list of attributes to be projected. The DURING interval attribute (renamed TEMP_DURING) is automatically included in the projection and does not need to be named in the attribute_list.

t_project {*attribute_list*} *relation_name*

The following query demonstrates the use of temporal project:

Query 4*: Every 90 seconds, return continuous periods during which any vehicle was present in a segment.*

```
DEFINE seg_Busy (segment, TEMP_DURING, DURING) AS
SELECT segment, TEMP_DURING, merge() as DURING
FROM
        90_second_timer nst,
        WINDOWED_SOURCE (nst, [0:0], {*} ,
                t_project {segment} segStr)
```

This example uses two input event streams: the segStr stream which is the output of Query 2, and the 90_second_timer. To retrieve continuous periods of time during which any car was present in any given segment, we perform a temporal projection of the segment attribute over segStr. The interval attribute is automatically included in the projection list and renamed to avoid a naming conflict with the window interval (i.e., the DURING attribute in the DEFINE statement). The temporal projection is then enclosed within a 90 second window. Finally, the segment number and the DURING interval resulting from the temporal projection (renamed TEMP_DURING) are included in the output. The window interval DURING is assigned as the output interval of the event. As an example consider the following data:

segStr

vehicleId	Segment	DURING
200	7	[091:102]
100	7	[100:105]
300	7	[095:107]
100	8	[106:110]
200	8	[103:112]
300	8	[108:116]
200	9	[113:120]
100	8	[115:120]
300	9	[117:124]

If segStr contains the above events, the result of temporal projection as reflected in seg_Busy is:

seg_Busy

Segment	TEMP_DURING	DURING
7	[091:107]	[091:180]
8	[103:120]	[091:180]
9	[113:124]	[091:180]

Where TEMP_DURING reflects the result of the temporal project operation and DURING is the interval timestamp of the associated window.

The behavior of the temporal projection operator is to aggregate overlapping events together when they have the same values for projected attributes. This can be useful in many situations that require distinct values of a subset of event attributes.

4.7.2 Temporal Intersection, Difference and Union

The set operations intersection, difference and union take two relations as input. The input relations need to have the same heading or, in other words, be union-compatible. The value of TEMP_DURING is calculated by first UNPACKing the relations, then performing a standard union, intersect or difference, and finally PACKing the result.

relation1_name t_intersect *relation2_name*
relation1_name t_difference *relation2_name*
relation1_name t_union *relation2_name*

The example queries below illustrate the use of each operator.

Query 6 (temporal intersection): Every 90 seconds, return periods during which cars with vehicle id equal to 100 and 200 were both in the same segment.

First we need to define two sub streams of segStr which correspond to two specific vehicles with identifiers equal to 100 and 200. The query for the vehicle with vehicleId=100 is:

```
DEFINE v100_segStr (segment, DURING) AS
SELECT
        vehicleId, segment, merge() as DURING
FROM
        segStr
WHERE
        vehicleId = 100
```

v200_SegStr is defined similarly as:

```
DEFINE v200_segStr (segment, DURING) AS
SELECT
        vehicleId, segment, merge() as DURING
FROM
        segStr
WHERE
        vehicleId = 200
```

Given the segStr events from Section 4.7.1 these queries produce the following results:

v100_segStr

Segment	DURING
7	[100:105]
8	[106:110]
8	[115:120]

v200_segStr

Segment	DURING
7	[091:102]
8	[103:112]
9	[113:120]

Now consider the intersection query below that again uses the WINDOWED_SOURCE and the 90_second_timer event:

```
DEFINE v100_I_v200 (segment, TEMP_DURING, DURING) AS
SELECT
        segment, TEMP_DURING, merge() as DURING
FROM
        90_second_timer nst,
        WINDOWED_SOURCE (nst, [0:0], {*},
                v100_SegStr t_intersect v200_SegStr)
```

The t_intersect operator returns periods of time during which v100_SegStr and v200_SegStr share all values and their intervals overlap with each other. The window specification projects the segment number and renames the interval attribute to avoid conflict with the window interval attribute. The query output consists of the segment number, the intersecting durations as TEMP_DURING, and the DURING output timestamp of the window.

v100_I_v200

Segment	TEMP_DURING	DURING
7	[102:105]	[091:180]
8	[106:110]	[091:180]

As can be seen, temporal intersection is useful for identifying when two interval-based event streams are ongoing during the same period of time. To know when one event was active, but not the other, the temporal difference operator can be used as shown in the next example.

Query 7 (temporal difference): Every 90 seconds, return periods during which car v100 was in any given segment and car v200 was not.

A difference query looks almost identical to an intersect query:

```
DEFINE v100_D_v200 (segment, TEMP_DURING, DURING) AS
SELECT
        segment, TEMP_DURING, merge() as DURING
FROM
        90_second_timer nst,
        WINDOWED_SOURCE (nst, [0:0], {*},
                v100_SegStr t_difference v200_SegStr)
```

The t_difference operator returns periods of time during which v100_SegStr was occurring, but a corresponding v200_SegStr, with the same segment number, was not. The query output consists of the segment number, the difference durations as TEMP_DURING, and the DURING output timestamp of the window.

v100_D_v200

Segment	TEMP_DURING	DURING
7	[103:105]	[091:180]
8	[115:120]	[091:180]

Conversely, we can state the following query:

Query 8 (temporal difference): Every 90 seconds, return periods during which car v200 was in any given segment and car v100 was not.

As:

```
DEFINE v200_D_v100 (segment, TEMP_DURING, DURING) AS
SELECT
        segment, TEMP_DURING, merge() as DURING
FROM
        90_second_timer nst,
        WINDOWED_SOURCE (nst, [0:0], {*},
                v200_SegStr t_difference v100_SegStr)
```

Which would yield the following result:

v200_D_v100

Segment	TEMP_DURING	DURING
7	[091:099]	[091:180]
8	[103:105]	[091:180]
8	[111:112]	[091:180]
9	[113:120]	[091:180]

As an example of Temporal Union, consider the following query:

Query 9 (temporal union): Every 90 seconds, return periods during which either car v100 or v200 were in any given segment.

```
DEFINE v100_U_v200 (segment, TEMP_DURING, DURING) AS
SELECT
        segment, TEMP_DURING, merge() as DURING
FROM
        90_second_timer nst,
        WINDOWED_SOURCE (nst, [0:0], {*},
                v100_SegStr t_union v200_SegStr)
```

The result of the temporal union query is:

v100_U_v200

Segment	TEMP_DURING	DURING
7	[091:105]	[091:180]
8	[103:112]	[091:180]
8	[115:120]	[091:180]
9	[113:120]	[091:180]

4.7.3 Temporal Restrict

Temporal restrict takes as input a relation and a series of restrict propositions that can be combined using AND and OR.

relation_name t_restrict (*restrict_propositions*)

The next example makes use of the temporal restrict operator. In general, temporal restrict is useful for specifying interesting points of time. Intuitively, a temporal restrict should be performed over interval-based attributes. If no interval-based attributes are included in the *restrict_propositions*, the result of temporal restrict would be no different from a regular restrict operation.

Query 10: (temporal restriction) Every 90 seconds, return the vehicle Id and segment numbers for the following points in time: 100, 110 and 120 .

```
DEFINE at100_110_120 (vehicleId, segment, TEMP_DURING, DURING) AS
SELECT
        vehicleId, segment, TEMP_DURING, merge() AS DURING
FROM
        90_second_timer nst,
        WINDOWED_SOURCE (nst, [0:0], {*},
        segStr t_restrict (TEMP_DURING=[100,100] OR
                           TEMP_DURING=[110,110] OR
                           TEMP_DURING=[120,120]))
```

This query uses the same time window as in the previous examples. The restrict condition is evaluated against the timestamp interval attribute of segStr, renamed TEMP_DURING. The result of the above query is:

at100_110_120

vehicleId	Segment	TEMP_DURING	DURING
200	7	[100:100]	[091:180]
100	7	[100:100]	[091:180]
300	7	[100:100]	[091:180]
100	8	[110:110]	[091:180]
200	8	[110:110]	[091:180]
300	8	[110:110]	[091:180]
200	9	[120:120]	[091:180]
100	8	[120:120]	[091:180]
300	9	[120:120]	[091:180]

It is important to note that it is not necessary for any single event in segStr to have an interval equal to [100:100], [200:200] or [300:300] in order to produce a non-empty result. Since the restrict operation occurs over the unpacked version of segStr, any event interval that simply includes any of these time points will be included in the result.

4.7.4 Temporal Join

A temporal join operator takes two relations and a list of attributes as input. This input is used to perform an equi-join. The DURING interval attribute (renamed TEMP_DURING) is automatically included in the JOIN operation and does not need to be explicitly listed in the attribute_list.

relation1_name t_join *relation2_name* on {*attribute_list*}

To demonstrate the temporal join operator, a smart office scenario is used. The following derived input event streams are available:

- empty_room (Room_id, dept, DURING): an event reflecting the periods of time during which a given room was empty.
- on_light (Room_id, DURING): an event reflecting the periods of time during which the light was on in a given room.

Assume the following data:

empty_room

Room_id	Dept	DURING
120	CSE	[091:115]
110	CSE	[125:160]
120	CSE	[130:170]

on_light

Room_id	DURING
110	[095:100]
110	[120:130]
120	[110:145]

Now consider the Query 11:

Query 11: (temporal join) Every 90 seconds, return events, reflecting the room_id, department names and continuous periods of time during which the room was empty and the light was on.

```
DEFINE emptyroom_lighton (room_id, Dept, TEMP_DURING, DURING) AS
SELECT
        room_id, Dept, TEMP_DURING, merge() as DURING
FROM
        90_second_timer nst,
        WINDOWED_SOURCE (nst, [0:0], {*},
                empty_room t_join on_light on {room_id})
```

The result of temporal join over the sample data is:

emptyroom_lighton

Room_id	TEMP_DURING	DURING
120	[110:115]	[091:180]
110	[125:130]	[091:180]
120	[130:145]	[091:180]

The functionality of temporal join is somewhat similar to temporal intersect, with the difference that temporal join can be applied to events with different headings, and the join condition can be specified by the query.

4.8 Summary

This chapter has presented the overall design and individual operators of TEQL. The overall language structure follows the select-from-where format of SQL queries. The attributes in the select list must include the timestamp attribute DURING. DURING is specified using the MERGE() function which allows control over the output timestamp value. The different event sources are provided in the FROM clause. A special kind of event source is the WINDOWED_SOURCE, which is used for specification of operators that have a non-monotonic behavior. The temporal relational operators RESTRICT, SELECT, UNION, INTERSECT, DIFFERENCE, and JOIN can be specified within a WINDOWED_SOURCE. The next chapter presents the relational framework used for formalizing TEQL and includes the specification of all TEQL operators.

Chapter 5

RELATIONAL FRAMEWORK AND LANGUAGE SPECIFICATION

The Composite Event Relational Algebra (CERA) is a relational framework developed to formalize the semantics of XChange^{EQ} (Eckert 2008). This chapter describes CERA and adapts it for formalizing the TEQL operators and language constructs. CERA is based on relational algebra. However, it imposes several limitations on relational algebra to achieve a property called *temporal preservation*. Temporal preservation enables continuous, step-wise evaluation of relational statements in an event environment. To support evaluation of TEQL operators over event streams, temporal relational operators must be defined to support temporal preservation.

After introducing CERA in Section 5.1, temporal preservation is discussed in Section 5.2. Timestamps form the basis for event-based queries, but are treated differently across event processing and temporal database frameworks. Section 5.3 presents some of these differences and explains the chosen characteristics for timestamps in TEQL. Section 5.4 formalizes different relational operators while upholding temporal preservation. This is achieved through the use of windows. The section includes formalization of the WINDOWED_SOURCE operator and the different temporal operators supported in TEQL. Section 5.5 formalizes the MERGE() function. Section 5.6 discusses temporal conditions. Section 5.7 presents a summary of the chapter.

5.1 CERA: A Relational Framework for Event Processing

CERA is a restricted version of relational algebra developed to model event queries (Eckert 2008). At a conceptual level, event histories are equivalent to relations. Each event type has a specific heading. Each data item of the event payload can be modelled as a typed attribute. Timestamps are also considered as interval typed attributes and are

included in relation headings as a mandatory attribute. Each arriving event is considered a tuple and is stored in the appropriate event history relation.

Relational algebra is a powerful tool to express queries, with operators that are properly defined and well understood. However, when applying relational algebra in an event setting, the fundamental differences between event histories and relational database tables need to be addressed. In a DBMS setting, the result of a query is calculated at the instant that it is run, using data that is already residing in the database. In an event environment:

- New event tuples arrive over time. At any given moment in time, only data that has arrived so far is available in the event histories to query.
- Queries run and produce new results continuously against an on-going stream of arriving event data

To illustrate this issue, we use the following example from (Eckert 2008):

> *Given incoming event streams 'order', 'shipped' and 'delivered', return completed orders, defined as an order event followed by a shipped event which itself is followed by a delivered event, with the order, shipped and delivered not being further than 48 hours from each other* .

To simplify discussion of the issues, a relational version of $XChange^{EQ}$ called Rel^{EQ} is introduced in (Eckert 2008). In Rel^{EQ} incoming events are assumed to be relational tuples instead of the XML format that is expected in $XChange^{EQ}$. The above query can be express in Rel^{EQ} as:

```
comp(id, p) ← o : order(id, p, q), s : shipped(id, t), d : delivered(t)
        o before s, s before d,
        {o, s, d} within 48
```

Translating the above query into relational algebra yields:

$$\sigma[\max\{o.e, s.e, d.e\} - \min\{o.s, s.s, d.s\} \leq 48]$$
$$(\sigma[s.e < d.s]($$
$$\sigma[o.e < s.s]($$
$$(R_o \bowtie S_s) \bowtie T_d)))$$

For ease of reading, instead of the traditional subscript notation, brackets are used for specifying the parameters of relational operators. The variables s and e are the beginning and ending timestamps of events, respectively.

While the above relational expression is a semantically accurate representation of the requested query, running it once will yield the correct results only if all relevant tuples from order, shipped and delivered are available. In other words, the above expression assumes full knowledge of every event that has happened in the past or will happen in the future. In reality, past events are available but future events are yet to come. In an event setting, the query needs to run continuously to accommodate for newly arriving data.

Continuously re-running a relational expression at every instant would yield the desired results for the above example. However, in general, continuously re-running a relational expression at every instant does not always lead to the desired solution. Specifically, one of the following problems might arise:

- Results generated at a specific step of a query might be incompatible with previously generated results, this is due to the existence of non-monotonic operators in the relational model
- A result simply might not be calculable over an infinite set. This is due to the existence of blocking operators in relational model.

If we consider the output of an operator as a relation itself, with input data being provided continuously over time, non-monotonic operators are operators that do not produce an append-only output. An example of such an operator is the difference operator. A tuple produced as a result of a query that involves a difference may need to be retracted at a

later point in time. In an event environment, the output of a query itself is considered a complex event which may be used in other event queries. Since event systems are usually employed over networks, physical retraction of a previous result is not possible.

Blocking operators are operators which require a closed set in order to produce a result. For instance, an aggregation operation such as SUM, cannot produce any output without access to the entire set that is to be aggregated. Since, in an event environment, the event histories are theoretically unbounded, any query involving an aggregation operation would block indefinitely.

Essentially, one of two approaches can be taken to address these two groups of operators. One approach is to develop a model that allows a logical retraction of previous results. In an event model, events are usually propagated over a network, possibly to different organizations. While physically removing previously generated results is not possible, there are ways to semantically invalidate previous results that disagree with new data. For instance CEDR (Barga and Caituiro-Monge 2006) allows modification of previously generated results, using the following mechanism:

- Each event is assigned a unique identifier
- Each event is assigned an extra timestamp assigned by the event provider, which reflects the validity interval for that event.

Invalidating a previously generated event consists of reproducing an event with the same identifier, but with an expired value for the validity interval. A new event is then generated to reflect the now current values and is assigned a valid interval.

The second approach is to restrict the semantics of the query language in such a way that at any given instant of time, *indefinitely correct* results can be calculated using available data only. In other words, events that arrive in the future should not affect query results generated at the current or previous instants of time.

TEQL adopts a model in which event queries can make use of events generated by other queries. In such a model, supporting retraction or invalidation of previous results can become complicated. Adjustments to previously generated output would need to take into account any other queries that might have used the invalidated events. The invalidation of results would need to be propagated upward through any existing query hierarchy. Due to this reason, TEQL follows the second approach, which is also adopted in CERA.

In this approach, use of blocking and non-monotonic operators is only allowed within time windows. The blocking and non-monotonic operators are evaluated on the set of tuples enclosed within the window and results are generated at the end of the window. In the following subsection, we look at *temporal preservation*, which is how CERA deals with these issues.

5.2 Temporal Preservation

The central property of CERA that enables evaluation of relational algebra-based queries in an event environment is called *temporal preservation*. Before discussing temporal preservation, we need to introduce the different relations possible in CERA and what constitutes the *occurrence time* for an event in each relation.

The conceptual relations within CERA can be broken into three groups, with different qualities:

- Primitive event histories – These relations store instantaneous events incoming from event sources.
- Complex event histories – These relations store the results of event queries, which are calculated based on different primitive events and/or other complex events

- Intermediate results – The result of any sub-expression of an event query can be viewed as an intermediate result.

Informally, the occurrence time for an event is the time at which the event was generated. For primitive events, which are instantaneous, the begin and end time are equal and, as a result, either one can be used as the occurrence time. For complex events, which may have a non-zero, interval-length, the occurrence time of the event is reflected in the end-time of the timestamp value. If multiple time-stamps are present, occurrence time of the result is the greatest end-time present in the event.

The temporal preservation property states that:

For any expression, current results (result with occurrence time equal to now) should be calculable using available data.

In other words, events and query results generated until this point in time should be sufficient for calculating any result belonging to the current point in time.

In CERA, the inclusion of timestamps in queries is restricted. Primitive events may only have a single timestamp attribute. An intermediate result may contain more than a single timestamp due to inclusion of several event sources within the query.

CERA only allows a single interval to be present in the final result of an event query. The different intervals that may be present in a query result due to the inclusion of multiple event sources are merged together using the merge operator, which will be discussed in Section 5.5.

To achieve temporal preservation, CERA restricts certain types of behaviour in relational algebra and also prohibits any modification of the timestamp values outside of the merge operator. This is because the timestamp of an expression is the only way of determining the occurrence time of a result, and is used for differentiating current results from previous ones (See Section 5.3).

In summary, the restrictions CERA imposes on relational algebra are:

- All relations must include at least one interval attribute
- Projection (π) cannot drop timestamp attributes
- The Grouping operator (γ) is a common extension of relational algebra that is used to offer a similar functionality to the SQL Group by clause. In CERA, γ is supported but it must use the begin and end time of all timestamp attributes as grouping attributes.
- Difference (\) and union ($\cup$) are not supported in CERA.
- A rule head must include a *merging* operator, which calculates the output timestamp interval so that its start time is equal to the minimum of input start times and its end time is equal to the maximum of end times. The merging operator also drops all other timestamp values.

These requirements together ensure that:

a) Non-monotonic or blocking operators are not allowed in an unrestricted way,
b) Timestamp values are not modified or omitted, and
c) The timestamp of the output of an event query spans the duration of all events participating in that result.

Collectively, these properties represent the Temporal Preservation property which enables incremental, step-wise evaluation of CERA expressions (Eckert 2008; François Bry and Eckert 2007).

5.3 Characteristics of Timestamps in TEQL

As mentioned previously, TEQL merges the semantics of temporal databases with event processing. A shared characteristic of temporal databases and event processing is the use of timestamps to signify the period of time during which the associated data item is considered to be valid. However, there are differences in the semantics and characteristics

of timestamps, both within and across the two frameworks. Before formalizing the different language features, these differences are examined to describe design choices that are made for semantics and characteristics of timestamps in TEQL.

Multiple, often widely differing, languages and/or proposals exist in both the event processing and temporal databases areas. Comparing all of these languages against each other would be a worthy endeavor, but would also prove to be quite lengthy and is beyond the scope and purpose of this dissertation. Here, we focus specifically on comparing the XChange^{EQ} event processing framework and the Date, Darwin and Lorentzos' approach to temporal databases. As proposals that are most directly relevant to the work in this dissertation in their respective research areas, they are particularly useful to motivate the semantics and characteristics of timestamps in TEQL.

A summary of the comparison can be found in Table 5.1.

	XChangeEQ framework		**Temporal Database framework**	
Timestamp	**CERA**	**XChangeEQ language**	**Valid Time**	**Transaction Time**
Count	Single	None	Single or Multiple	Single
Optional/ Mandatory	Mandatory	N/A	Optional Non-temp DB if excluded	Optional
Visibility	Visible	Hidden	Visible	Visible
Manipulation	Automatically assigned	N/A	Can be manipulated	Automatically assigned

Table 5.1. Timestamps in XChange^{EQ} and Date et al.'s Temporal Database Framework

For the XChange^{EQ} framework, the XChange^{EQ} query language and CERA are both examined. Valid time and transaction time are considered for temporal databases. The concepts of valid time and transaction time were presented in Section 3.1.

The comparison is made across the following dimensions:

- Count – The number of timestamps or interval attributes allowed in each approach.
- Optional/Mandatory – Whether or not inclusion of the timestamp is required.
- Visibility – Whether or not the timestamp attribute is visible to query.
- Manipulation – Whether or not the timestamp value can be modified.

The CERA framework requires a single timestamp to be assigned to primitive and complex events. Intermediate results may have multiple timestamp attributes. These are consolidated into a single timestamp attribute using the merge operator when generating the query output as a complex event. No direct manipulation of the timestamp value is possible in CERA in order to uphold temporal preservation. The timestamp is not visible at the XChange^{EQ} query language level and consequently cannot be manipulated through the use of the query language either. When describing temporal conditions, for instance using Allen's operators, it is the events themselves that are referenced. Consider the example from the beginning of this chapter: An order event (o) happening before a shipped event (s) and a shipped event happening before a delivered (s) event, all within 48 hours. This is expressed in XChange^{EQ} as:

```
o before s, s before d,
{o, s, d} within 48
```

As can be seen, no reference is made (nor can be made) to the timestamps. It is the events themselves that participate in these conditions and are said to hold these temporal relationships.

In the temporal database framework, there is no restriction on the number of interval type attributes that may be present in a relation. Each interval may have a separate meaning, and signify the validity of a certain proposition. In the strictest sense,

only a single interval attribute would be needed to reflect the validity of the entire tuple. Any additional tuples reflecting the same interval would be redundant. Other interval typed attributes may be present to reflect other data of the interval type.

The transaction time is a single attribute maintained by the DBMS. It is not necessary to have a transaction time stored for a database or even a temporal database. It is only needed if the database is used to store historic data besides current data. In this case, the DBMS automatically assigns the transaction time interval to reflect the period during which each tuple was present in the historic and current states of the database.

Not being restricted to a single interval attribute can be useful in an event setting. For instance consider again the example of the order delivery system. Besides the timestamp of the event, the order may include an interval attribute which reflects when the receiving party would be available for the item to be delivered. Alternatively, consider a smart office scenario in which events are sent out every five minutes regarding the status of sensors and equipment within a room. Such an event could contain data such as periods of time during which the room temperature fell below a predefined threshold or when a light was on. In both cases, these non-timestamp intervals are temporally relevant and could be meaningfully combined with other temporal data, including the timestamp itself, to derive new information. There seems to be no clear reason to omit a data item for the reason that is of a temporal type.

In CERA, the value for the timestamp is automatically assigned. For primitive events, this value is assumed to be assigned either by the event provider or through a global clock. For complex events, the timestamp is assigned through the mandatory inclusion of the merge operator in the rule heading. At the XChange^{EQ} language level, assignment of the interval through the merge operator is hidden. The interval value cannot be modified directly by the query.

In the temporal database model, given the proper privileges, all data, and meta-data are visible and can be queried. This is due to the basic underlying principle in the relational model, known as the *information principle*. As a result, both the valid time and the system generated transaction time can be queried by the user. The value of the valid time can also be modified, within any imposed constraints. The value of the transaction time, however, is not directly modifiable by the user, as it would create inconsistencies between historic states of the DBMS.

The reasoning for the strict control on the assignment of the timestamp value in XChange^{EQ} is that the value of the timestamp determines the occurrence time of the event. The occurrence time of the event is the basis for determining when an event was produced, which is critical in the correct execution of the query evaluation cycle. This is why in the CERA definition modification of the output timestamp outside of the merge operator is considered a violation of temporal preservation.

TEQL, is also dependent upon temporal preservation for correct evaluation of event queries. However, TEQL aspires towards the more open approach of temporal databases as much as possible, which results in more expressive power for the query. In this light, while acknowledging that some restriction on the value of the output timestamp is probably necessary, we reexamine the conditions that CERA places on the output timestamp value:

- The begin time of the output timestamp does not affect the occurrence time of the event in anyway, so modification of it should not affect temporal preservation.
- Pushing the end time of an event into the past is indeed problematic as it would imply that other queries which were being evaluated at the time, did not consider this event at the correct time.

- Pushing the end time of an event into the future, does not cause such a problem. The query processor could watch for such events, and only release them after an appropriate delay at the correct time.

In light of these possibilities, TEQL allows modification of the output timestamp. This capability will be discussed further in Section 5.5.

In summary TEQL has the following characteristics for timestamps across the previously presented dimensions:

- Count – TEQL allows for multiple interval attributes to be present in primitive and composite events. However, a specific interval attribute named DURING is used for the timestamp value and is the basis for determining an event's occurrence time.
- Optional/Mandatory – Inclusion of DURING is mandatory in every primitive and composite event. Intermediate results must include the values of DURING attributes for all underlying event queries. These will be renamed to <*source_name*>_DURING to avoid conflict with each other, where <*source_name*> is the name of the corresponding event. Other interval attributes may be included or omitted as desired.
- Visibility – Following the information principle, all event attributes including interval attributes in general and the timestamp specifically are visible and can be queried.
- Manipulation –The value of <*source_name*>_DURING attributes may not be manipulated in intermediate results. Manipulation of the output interval is possible. However, it is restricted in that the query can only manipulate the value of DURING in the output using the merge function (see Section 5.5).

The remainder of this chapter formalizes the different operators of TEQL. This formalization is based on CERA. When necessary, CERA is extended to support the different features in TEQL. After presenting the different language constructs, a summary of extensions to CERA is provided at the end of the chapter in Section 5.7.

5.4 Temporal Relational Operators

This section formalizes the evaluation of the temporal relational operators in an event setting. These operators were presented in Section 3.2.3 in the temporal database context. To evaluate temporal operators in the context of event processing, the PACK and UNPACK operators are needed. Since these operators do not appear originally in CERA, the operators need to be added to CERA. Evaluating PACK and UNPACK in an event setting is discussed in Section 5.4.1. This section also explains the need for using windows when using temporal relational operators. The WINDOWED_SOURCE operator, which is designed for this purpose, is formalized in Section 5.4.2. Section 5.4.3 formalizes each of the temporal relational operators.

5.4.1 Evaluating PACK and UNPACK Over Event Streams

Non-monotonic or blocking operators cannot be directly applied in an event setting, hence these qualities should be examined for PACK and UNPACK. In this section, the temporal project example is used to discuss the monotonicity and blocking properties for PACK and UNPACK. The example in Figure 5.1 illustrates temporal project of the Dept name over the Employee_Dept relation.

(A) Employee_Dept

id	Dept	DURING
e1	CSE	[1002:1003]
e2	CSE	[1003:1004]
e3	ECE	[1006:1007]

(B) Unpacked Employee_Dept

id	Dept	DURING
e1	CSE	[1002:1002]
e1	CSE	[1003:1003]
e2	CSE	[1003:1003]
e2	CSE	[1004:1004]
e3	ECE	[1006:1006]
e3	ECE	[1007:1007]

(C) Project Dept, DURING

Dept	DURING
CSE	[1002:1002]
CSE	[1003:1003]
CSE	[1004:1004]
ECE	[1006:1006]
ECE	[1007:1007]

(D) Pack

Dept	DURING
CSE	[1002:1004]
ECE	[1006:1007]

Figure 5.1 Temporal Project of Dept over Employee_Dept

Temporal project is broken down into three steps: UNPACK, PROJECT and PACK. Assume that the data for the Employee_Dept relation arrives gradually over an event stream and in the order of ending timestamp value. Accordingly, the operations corresponding to the three steps are also evaluated with the arrival of new data, and the output relations are updated.

UNPACK:

At time 1003, the following tuple is inserted into Employee_Dept:

E1, CSE, [1002, 1003],

Unpacking Employee_Dept yields:

E1, CSE, [1002, 1002],
E1, CSE, [1003, 1003],

At time 1004 the following tuples have been inserted into Employee_Dept:

E1, CSE, [1002, 1003],
E2, CSE, [1003, 1004]

Now unpacking Employee_Dept yields.

E1, CSE, [1002, 1002],
E1, CSE, [1003, 1003],

E2, CSE, [1003, 1003],
E2, CSE, [1004, 1004]

As can be seen, the result of UNPACK for a relation is equivalent to the UNION of unpacking all its tuples. UNPACK can be applied to each individual tuple as it arrives, independent of other tuples. Clearly UNPACK is not a blocking operator.

If a newly unpacked tuple already exists in previously unpacked tuples, the duplicate tuple will not be added. Since unpacking the entire relation all at once or unpacking each tuple as it arrives, produces the same result, it is clear that tuples unpacked at a later time, do not affect previously unpacked tuples. Hence, the resulting relation from an UNPACK operation is append only. As a result UNPACK, is a monotonic operator.

PACK:

At time 1003, the result of projection would consist of the following tuples:

CSE, [1002 : 1002],
CSE, [1003 : 1003]

The result of PACK would be:

CSE, [1002 : 1003]

At time 1004 the result of projection would be:

CSE, [1002 : 1002],
CSE, [1003 : 1003],
CSE, [1004 : 1004]

The result of PACK would be:

CSE, [1002 : 1004]

This result is inconsistent with the result of PACK at time 1003. As more tuples arrive in the input, the previous output of PACK is no longer correct and needs to be replaced with the current result. Therefore applying PACK at each step produces a result that can only

be guaranteed to be correct for that instant in time. This makes the PACK operation non-monotonic.

Compared to aggregate functions PACK is similar to calculating a SUM function. Similar to SUM, over a closed set of tuples, PACK yields the same final result, regardless of whether the closed set was passed to the PACK operation at once or gradually. With PACK, tuples generated at a later point in time may have equal data values with a tuple generated at an earlier point in time. In this case, if the intervals of the two tuples overlap, they will be merged together into a single tuple. The interval for the resulting tuple would encompass both intervals, and neither of the tuples with overlapping intervals would appear in the result. This means that the previously generated tuple would need to be retracted. Thus, PACK can only generate a definite result when applied to a closed set of tuples. In this sense PACK is a blocking operation.

As discussed previously, all temporal relational operators are based on the following three steps:

- UNPACK
- [regular] Relational Operation
- PACK

While a few of the temporal relational operations do not require the initial UNPACK step, all of them require the PACK step. Since PACK is non-monotonic, all temporal relational operators are also non-monotonic. As a result temporal relational operators need to be defined within a window if they are to be used in an event processing environment.

To formalize the temporal relational operators, we add the PACK and UNPACK operators to the relational algebra. The following notation will be used to express the PACK and UNPACK operators:

PACK:

P [*Interval_Attribute*] (*Relation_Name*)

UNPACK:

N [*Interval_Attribute*] (*Relation_Name*)

where *Relation_Name* is the name of the relation, and *Interval_Attribute* is the name of an attribute of interval type in *Relation_Name.*

Since PACK and UNPACK modify the value of the timestamp, using them in an unrestricted way would violate temporal preservation.

To uphold temporal preservation, the WINDOWED_SOURCE operator is used. The window interval becomes the timestamp, or occurrence duration, for the window and consequently for each event that is produced by the window. WINDOWED_SOURCE renames the timestamp of the input event to TEMP_DURING before PACK and UNPACK are applied to uphold temporal preservation.

5.4.2 The WINDOWED_SOURCE Operator

The purpose of a window operator is to enable non-monotonic operators in an event setting, providing concrete boundaries over which such operators can be evaluated. In the WINDOWED_SOURCE operator this functionality is supported, while additionally providing control over window boundaries. The result of the WINDOWED_SOURCE is a relation which participates in the rest of the query. As presented in Section 4.3, in TEQL windows are specified using the WINDOWED_SOURCE operator:

WINDOWED_SOURCE((*interval_source_alias*, [*b_offset*, *o_offset*]), {*window_attributes*}, [*temporal_ operation*|*aggregate_operation*])

The WINDOWED_SOURCE operator consists of the following sub-operations:

- Setting the value of the window boundaries: An event alias, *interval_source_alias*, is taken as input. This event alias must be defined in the query that is associated with the window and forms the basis for the window

boundaries. WINDOWED_SOURCE allows integer valued offsets, *b_offset* and *o_offset*, to be applied to the window boundaries.

- Applying the window to the non-monotonic operation: The non-monotonic operation is encapsulated within the window boundaries through the use of a special join operation, called a *window join*, which will be discussed shortly.
- Preserving original timestamp values: The window boundaries form the occurrence duration, or DURING value for the window. To support temporal preservation and incremental query evaluation, timestamps of events participating in the underlying query need to be preserved separately. This is done through renaming the timestamps of the events in the underlying query to TEMP_DURING, before applying the window and performing the specified operation.
- Projection of attributes: The WINDOWED_SOURCE operator allows a list of attributes, the *window_attributes*, to be projected.

Each of these steps is formalized and discussed below:

Setting the value of the window boundaries:

In XChange^{EQ}, a relative timer event which is specified separately can be used to create and modify the window boundaries. In TEQL, the WINDOWED_SOURCE operator can directly modify the window boundaries. Modification of a window is defined similar to the specification of relative time events in XChange^{EQ} as follows (Eckert 2008):

$$WB := \{ x \mid (x(o.b),x(o.e)) \in \pi[o.b, o.e](IS_o),\ x(w.b) = x(o.b) + b\_offset,\ x(w.e) = x(o.e) + e\_offset \}$$

where IS is the interval source, o is the alias for IS, and w is the alias for the output window boundary relation WB. Essentially calculating the window boundary relation is done by applying the b_offset and e_offset values to the interval attribute of the interval source relation, IS. The result becomes the DURING attribute for the windows source.

Applying the window to the non-monotonic operation

In CERA an operator, called a *temporal θ join*, is used to formalize windows over aggregation operations. The word "temporal" in the temporal θ *join* is not related to the temporal database operators as discussed in this dissertation or in the temporal database literature. To prevent any confusion, we will refer to this operator as a *window join*. A window join is defined as (Eckert 2008):

$$W \bowtie_{i \sqsupseteq j} R = \sigma\,[i.s \leq j.s \wedge j.e \leq i.e](W \bowtie R)$$

where i and j are occurrence durations for W and R respectively, and i.s, j.s and i.e, j.e refer to their start and end timestamps, respectively. The notation i⊒j is used to signify the falling of one interval (j) within the other (i) as expanded in the selection condition above. The result includes tuples of W joined with R, but only includes those tuples where R's interval falls within the interval of a corresponding W window. This creates a windowing behavior over R with the interval of W serving as the bounds of the window. As can be seen, the window W is also an event relation. Any defined event in the system can be used as a window. In TEQL, W is specified using the WINDOWED_SOURCE operator.

Besides ensuring that tuples from the input event (R) fall within the bounds of the window (W), a window join has a side effect. The window join tags the tuples of R with additional attributes from the window (W). These additional attributes can be any number of attributes from the event that the window is based on, but at the very least must include the DURING interval attribute of W.

Preserving original timestamp values

The aggregation itself is handled through the grouping operator γ in CERA. The γ operator is an extension to relational algebra with behavior similar to the SQL group by clause. The grouping operator is specified as follows:

$\gamma[a_1,\ldots,a_n, a \leftarrow F(A)](R)$

where $a_1,\ldots,a_n$ represent the grouping attributes and F(A) represents the aggregation operation. To uphold temporal preservation, the following conditions are enforced when evaluating aggregate operations over windows (Eckert 2008):

- Since aggregation operators, such as SUM or AVG, are blocking operators, they are only allowed over a window
- The timestamp of the window must be one of the grouping attributes. This ensures that only items falling within the same window can be aggregated together, and the value of the window boundaries is preserved in the aggregation. As a result, temporal preservation is supported.

In TEQL the formalization of aggregation operations is identical to CERA. However, temporal relational operations require renaming the timestamp value before being applied.

To illustrate the renaming of the timestamp value before evaluating the underlying query and application of the window join, consider the example of a temporal project. To evaluate a temporal project, the following operations are performed:

First, DURING is renamed to TEMP_DURING in the relation that temporal project is to be performed over (R).

1. $R_1 = \rho[TEMP\_DURING/DURING](R)$

In the second step the window join operation is performed.

2. $R_2 = (W \bowtie_{i \sqsupseteq j} R_1)$

Here W is the *interval_source_alias* after its timestamps have been altered by the offsets. In the next step, the UNPACK operation is performed over TEMP_DURING:

3. $R_3 = N\ [TEMP\_DURING]\ (R_2)$

Now the actual PROJECT operation is performed. The attributes provided in *project_list* are projected, along with DURING and TEMP_DURING.

4. $R_4 = \pi$ [*project_list*, TEMP_DURING, DURING] (R_3)

Finally the result is packed on TEMP_DURING:

5. TR = P [TEMP_DURING] (R_4)

The resulting relation R_5, will contain all the attributes specified in the *project_list* for the temporal project, along with two interval attributes: DURING and TEMP_DURING. DURING is the occurrence time of the window, and TEMP_DURING contains the interval which is formed as the result of performing the temporal project.

The five steps above can be demonstrated in a single formula as:

TR=P[TEMP_DURING]
(π [*project_list*, TEMP_DURING, DURING]
(N [TEMP_DURING]
(W $\bowtie_{i \sqsupseteq j}$ (ρ[TEMP_DURING/DURING](R)))))

All temporal relational operators are evaluated after application of the window. The formula for evaluating each temporal relational operator is presented in Section 5.4.3. However, the application of the window is omitted from each operator. In other words, the temporal operators are formalized over operands that are assumed to have already been subjected to the window join operator.

Projection of attributes:

As a final step, before being presented to the rest of the parent query, WINDOWED_SOURCE allows the list of output attributes to be defined:

WS = π[*window_attributes*, TEMP_DURING, DURING](TR)

TEMP_DURING and DURING must be included in the result of WINDOWED_SOURCE.

5.4.3 Specification of Temporal Relational Operators

Given the fact that temporal relational operators in TEQL are evaluated over windows, formalizing each operator becomes straightforward. In this section, the algebraic formulae for the different temporal operators are presented, and any simplifications or special cases are discussed. In all cases, it is assumed that the operand relations have already been subjected to the window join and contain a TEMP_DURING attribute.

Union, Intersect and Difference

These three binary relational operators are similar to each other. All three operators require that both operands have the same heading, or in other words be union-compatible. This heading should include an interval type attribute called TEMP_DURING.

Temporal intersect:

relation1_name t_intersect *relation2_name* ≡
P [TEMP_DURING] ((N[TEMP_DURING](*relation1_name*) ∩ N[TEMP_DURING](*relation2_name*))

where ∩ is the intersection operator as defined in relational algebra.

Temporal difference:

relation1_name t_difference *relation2_name* ≡
P [TEMP_DURING] ((N[TEMP_DURING](*relation1_name*) \ N[TEMP_DURING](*relation2_name*))

where / is the difference operator as defined in relational algebra.

Temporal union:

relation1_name t_union *relation2_name* ≡
P [TEMP_DURING] ((N[TEMP_DURING](*relation1_name*) ∪ N[TEMP_DURING](*relation2_name*))

where ∪ is the difference operator as defined in relational algebra.

Since a temporal union does not require the unpack step, it can be simplified as:

relation1_name t_union *relation2_name* ≡
P [TEMP_DURING] ((N[TEMP_DURING](*relation1_name* ∪ *relation2_name*))

Restrict and Project

Temporal restrict:

relation_name t_restrict ([*p*]) ≡ P[TEMP_DURING]
(σ [*p*] (N[TEMP_DURING](*relation_name)*))

where σ is the restrict (select) operator from relational algebra. *p* is the restrict condition which is provided in the form of a propositional formula.

Temporal project:

t_project {*attribute_list*} *relation_name* ≡ P[TEMP_DURING]
(π [*attribute_list*, TEMP_DURING, DURING]
(N[TEMP_DURING](*relation_name)*))

where π is the standard projection operator from relational algebra and *attribute_list* is the list of attributes to be projected. DURING and TEMP_DURING are automatically included among the projected attributes and do not need to specified by the user.

Since temporal project does not require the input relation to be unpacked it can also be formulated, more simply, as:

t_project {*attribute_list*} *relation_name* ≡ P[TEMP_DURING]
(π [*project_list*, TEMP_DURING, DURING] (*relation_name*))

Join

relation1_name t_join *relation2_name* on {*predicate_list*} ≡ P [TEMP_DURING] (
(N[TEMP_DURING](*relation1_name*) $\bowtie_{\varphi}$ N[TEMP_DURING](*relation2_name*))

To support more expressive conditions, the join operator here, $\bowtie_{\varphi}$, is the θ-join operator, rather than the natural join, as defined in relational algebra. θ-join can be defined as:

$$relation1\_name \bowtie_{\varphi} relation2\_name \equiv \sigma_{\varphi}(relation1\_name \times relation2\_name)$$

where × is the Cartesian product and φ is a list of predicates. φ includes the predicate_list provided in the t_join operator with the following predicate added:

relation1_name.TEMP_DURING = *relation2_name*.TEMP_DURING

5.5 The Merge Function

As defined in CERA, given a set of input intervals $i_1,\ldots,i_n$, the merge operator, μ, calculates an output interval *j*, such that:

$$j.s = \min \{ i_1.s, \ldots , i_n.s \} \text{ and}$$
$$j.e = \max \{ i_1.e, \ldots , i_n.e \}$$

The merge operator can be formulated using relational algebra as (Eckert 2008):

$$\mu[j \leftarrow i_1 \sqcup \ldots \sqcup i_n](E) \equiv \pi [\, j.s \leftarrow \min \{ i_1.s, \ldots , i_n.s \},\ j.e \leftarrow \max \{ i_1.e, \ldots , i_n.e \},\ sch(E) \setminus \{ i_1.s, \ldots , i_n.s, i_1.e, \ldots , i_n.e \} \,](E)$$

Essentially, the standard merge operator of CERA does two things:

- It calculates a new output timestamp value, *j*, which encompasses the value of all input timestamps
- All input timestamps are omitted from the output schema leaving *j* as the only interval present in the output event. *sch*(*E*) is the schema of the input event. *sch*(*E*) \ { $i_1.s, \ldots , i_n.s, i_1.e, \ldots , i_n.e$ } omits all preexisting interval attributes from *E*.

The MERGE function in TEQL, which we denote as MERGE() with parenthesis to differentiate from the CERA merge operator, shares its default basics semantics with the CERA merge operator. However, MERGE() differs from the CERA merge operator in the following ways:

- MERGE() takes as input two operands b_offset, e_offset which allow for the manipulation of the output timestamp value. The manipulation of the ouput timestamp will be discussed in detail below.
- The output of MERGE() is stored in the attribute name DURING.
- MERGE() does not omit existing timestamp values.

To represent MERGE() we define μ′ as:

$$\mu' [\text{DURING} \leftarrow i_1 \sqcup \ldots \sqcup i_n, b\_offset, e\_offset](E) \equiv$$
$$\pi [\ \text{DURING}.s \leftarrow (\min \{ i_1.s, \ldots, i_n.s \}) + b\_offset,$$
$$\text{DURING}.e \leftarrow \max \{ i_1.e, \ldots, i_n.e \} + e\_offset,$$
$$sch(E)\ ](E)$$

where b_offset and e_offset are optional integer parameters to MERGE(). If offset values are not provided, they are both assigned a value of zero, and MERGE() produces the same output interval as the CERA merge operator.

When a query involves only a single input, the output of the MERGE() function would be the same as the input event after application of any given offsets. Examples of the MERGE() function can be found in Sections 4.4 and 4.5.

The b_offset can be assigned any integer value. Assigning a negative value to b_offset causes the output interval to become longer by pushing the start time back into the past. Assigning a positive value to b_offset causes the output interval to become shorter by pulling the start time forward into a future point in time, with respect to the original start time.

The e_offset can only be assigned positive values, which cause the output interval to become longer by pushing the end time into the future. If negative values were allowed for e_offset, the end time of the output interval could also be pushed into the past. The end time of the output interval is also the occurrence time of the event. Allowing a decrease in the occurrence time of an event output causes the event to be generated at a later time than its occurrence time. This violates temporal preservation, and would cause already generated results of other event queries that make use of this event to become inaccurate.

For example, if MERGE could specify an e_offset equal to -5, that would mean that an event output with occurrence time 1000 would be generated at 1005, in other words the event would be arriving late. This makes results of event queries using the late event inconsistent. At time 1005 event queries running on top of this late event would

have already generated results pertaining to time 1000, 5 time units ago, without considering this late event.

Providing a positive value for e_offset can be used to shift the begin time of an interval into the future. However, this is limited by the value end time produced for the output timestamp since the begin time of the output interval is required to be less than or equal to the end timestamp. For instance, if before applying the offsets, the interval output is equal to [1000 , 1002] and b_offset is equal to +5, applying the b_offset unconditionally would result in an invalid interval [1005 , 1002] where the begin time is greater than the end time.

In the absence of any offsets, it should be clear that, if $i_1, \ldots, i_n$ are input intervals:

$$\min \{ i_1.s, \ldots, i_n.s \} \leq \max \{ i_1.e, \ldots, i_n.e \}$$

This is a logical consequence of the fact that for each $i_k \in \{ i_1, \ldots, i_n \}$; $i_k.s \leq i_k.e$ However, after applying the offsets the sanity assumption cannot be guaranteed, and the begin time of the output interval might become greater than the end time. To avoid producing an invalid interval, instead of a simple addition, the begin time is calculated as follows:

$$\text{DURING}.s = \begin{cases} \min \{ i_1.s, \ldots, i_n.s \}) + b\_offset \; iff \; \min \{ i_1.s, \ldots, i_n.s \}) + \\ \qquad b\_offset \leq \max \{ i_1.e, \ldots, i_n.e \}) + e\_offset \\ \\ \max \{ i_1.e, \ldots, i_n.e \}) + e\_offset \; iff \; \min \{ i_1.s, \ldots, i_n.s \}) + \\ \qquad b\_offset > \max \{ i_1.e, \ldots, i_n.e \}) + e\_offset \end{cases}$$

Effectively, this means that if, after application of the offset values, the calculated value for the start time becomes greater than the end time of the output interval, the start time is assigned the same value as the end time, and the output event becomes a zero-length, instantaneous event.

Another way to resolve this issue would have been to push the end time of the event into the future. However, as a design choice this would make the occurrence time of the generated event inconsistent with the user-defined value. Since the end time is always calculated as requested, in all cases the language produces events which are consistent with the user defined occurrence time.

5.6 Temporal Conditions

In TEQL, temporal conditions can be specified directly over timestamp values or through operators based on Allen's relationships. Allen's relationships are in reality just a shorthand since they can be directly specified in terms of conditions over timestamps also. In TEQL the temporal conditions are simply relational selection conditions over timestamp values.

5.7 Summary

This chapter has presented a relational framework for evaluation of event queries. The relational framework is based on CERA, with extensions to support the new temporal functionality possible in TEQL. CERA imposes an important quality called temporal preservation over standard relational algebra. Temporal preservation disallows non-monotonic behavior and limits modification to timestamp values. The different kind of restrictions that are placed on temporal values in XChange^{EQ}, CERA, and valid and transaction timestamps in Date et al.'s temporal database framework, were discussed. TEQL timestamps have the following qualities:

- Multiple interval attributes are allowed. A specific interval attribute named DURING is required to store the event's occurrence time.
- Every timestamp value can be directly queried in the language.
- Interval attributes other than DURING can be directly manipulated. DURING can only be manipulated in a controlled manner to uphold temporal preservation.

Each language operator was formalized using the extended CERA framework. In summary the following changes were made to the CERA framework, while upholding the temporal preservation property:

- Added support for PACK and UNPACK
- Added the WINDOWED_SOURCE operator for non-monotonic behavior. Because of their reliance on PACK, temporal relational operators can only be evaluated in the context of a window join.
- Redefined PROJECT, RESTRICT, UNION, INTERSECT, DIFFERENCE and JOIN in the context of temporal relational operators
- Added temporal conditions involving time points
- Added a MERGE() function that allows manipulation of the output timestamp value through specification of offsets.

Chapter 6

INCREMENTAL EVALUATION OF TEQL

A query plan-based approach is more flexible than the automata-based model, but it also presents some challenges for efficient query processing of event queries. The event pattern evaluation structure used in automata-based approaches directly reflects the sequential nature of incoming events. Query plans, in contrast, are applied to relations and do not natively represent any temporal order. The relational approach to query processing is based on the assumption that all tuples required for processing of the query are currently residing in the underlying relations. This assumption obviously does not hold in the event processing context; in reality, the input event streams are continuously generating new events and the corresponding event histories get updated accordingly. Therefore, the query plan-based approach needs to be adapted to support the continuous arrival of new events.

This chapter discusses the incremental evaluation approach used in TEQL to address this issue. The approach is based on the work in (François Bry and Eckert 2007) with extensions made to support TEQL functionality for temporal relational operators and the MERGE function. Section 6.1 discusses the concepts of incremental evaluation developed for CERA. Incremental evaluation is made possible through the use of a technique called *finite differencing*. Section 6.2 discusses the finite differencing of the different operators used in TEQL. Finally, Section 6.3 provides an example of finite differencing for a temporal query expression, to illustrate the process that needs to takes place for all TEQL queries before they can be evaluated in a step-wise incremental manner.

6.1 Incremental Evaluation of CERA

CERA expressions assume full knowledge of every event that has happened in the past or will happen in the future. In reality, access to past events is possible through the event histories, but future events are yet to come and thus cannot be accessed. In contrast to the standard one-time evaluation of relational queries, relational expressions used to express event queries need to be evaluated in a continuous, step-wise manner.

One approach to cope with the continuously arriving data is to reevaluate the entire query plan tree in each evaluation step. Using this approach, as new events arrive, correct and continuous results can be generated in the query output. However, as noted in (Eckert 2008), this approach is far from efficient. Essentially, in each step, what is being computed is not only the output of the query based on the new events, but also all the results corresponding to previous steps. The redundant, previously computed results need to be omitted through a select operation, which would filter results pertaining to previous steps. As time goes by, the set of results that are recomputed keeps on growing and performance continuously degrades. Furthermore, this approach is wasteful because it first computes unneeded results, which then need to be identified and omitted.

Ideally, it is desirable to calculate at each step, only the new results that are generated as a result of new input data.

XChange^{EQ} (Eckert 2008) addresses the problem of re-computation of previously computed results through a technique called *finite differencing*. In this approach, for each evaluation step, the input data is categorized into two distinct sets:

- The past data, designated with o: This is data that existed in the relevant histories prior to the current instant of time,
- The delta, designated with Δ: This is the new data in the relations based on the arrival of new events. The delta consists of the newly arrived primitive events,

for the primitive event histories, and the resulting tuples in the relational expressions for any materialized intermediate results.

Using finite differencing, CERA expressions associated with each query, and materialized intermediate results, are rewritten based on the o and Δ operators. In the rewritten form, queries do not make unqualified references to event histories, rather, each query takes as input the past data and the deltas. For instance given a join operation E1 ⋈ E2, the delta of a join operation can be calculated using the following equation:

$$\Delta(E1 \bowtie E2) = \Delta E1 \bowtie oE2 \cup \Delta E1 \bowtie \Delta E2 \cup oE1 \bowtie \Delta E2$$

Clearly, any result of joining the new data in E1 and E2 would also be a new result for the JOIN operation. If, for instance, E1 has received new data, it also needs to be joined with E2's old data (and vice versa). Hence, the union of the three joins forms the delta of the JOIN operation. The history of a join operation is calculated as:

$$o(E1 \bowtie E2) = oE1 \bowtie oE2$$

In contrast, with the re-computation approach, using the equation given for Δ of JOIN, calculating oE1 ⋈ oE2 at every step is avoided. Finite differencing for the rest of the CERA operators are specified in Figure 6.1 (Eckert 2008).

$\Delta\ \sigma\ (E) = \sigma\ (\Delta E)$	$o\ \sigma\ (E) = \sigma\ (oE)$
$\Delta\ \rho\ (E) = \rho\ (\Delta E)$	$o\ \rho\ (E) = \rho\ (oE)$
$\Delta\ \pi\ (E) = \pi\ (\Delta E)$	$o\ \pi\ (E) = \pi\ (oE)$
$\Delta\ \mu\ (E) = \mu\ (\Delta E)$	$o\ \mu\ (E) = \mu\ (oE)$
$\Delta\ \gamma\ (E) = \gamma\ (\Delta E)$	$o\ \gamma\ (E) = \gamma\ (oE)$
$\Delta\ (E_1 \cup E_2) = \Delta E_1 \cup \Delta E_2$	$o\ (E_1 \cup E_2) = oE_1 \cup oE_2$
$\Delta\ (E_1 \bowtie E_2) = \Delta E_1 \bowtie \Delta E_2 \cup oE_1 \bowtie \Delta E_2 \cup \Delta E_1 \bowtie oE_2$	$o\ (E_1 \bowtie E_2) = oE_1 \bowtie oE_2$
$\Delta\ (E_1 \bowtie_{i \sqsupseteq j} E_2) = \Delta E_1 \bowtie_{i \sqsupseteq j} oE_2 \cup \Delta E_1 \bowtie_{i \sqsupseteq j} \Delta E_2$	$o\ (E_1 \bowtie_{i \sqsupseteq j} E_2) = oE_1 \bowtie_{i \sqsupseteq j} oE_2$

Figure 6.1 Finite differencing of CERA operators

The operator parameters have been removed for easier readability. Note that these finite differencing formula only apply to the restricted versions of relational operators in CERA and may not hold for the corresponding standard relational operators.

After finite differencing of relational expressions for all given queries, a step-by-step algorithm is used to continuously calculate the new results for each query. The algorithm for evaluating the finite differenced queries is as follows:

Given a query of the form $Q_n := E_n$, where Q_n indicates a query or a materialization point name, and E_n the corresponding relational algebra expression, the aim is to calculate ΔQ_n using the following input items (Eckert 2008):

1. *The relations that contain the newly arrived primitive events* $\Delta B_1, \ldots, \Delta B_m$. If B_i is the virtual event history relation that includes all events that ever occurred, be it in the past, present or future, ΔB_i is those events whose maximum end time stamp, or occurrence time is equal to *now*. Formally: $\Delta B_i = \sigma\,[M_{Bi}=now](B_i)$ where M_{Bi} is the maximum end timestamp value in B_i

2. *Base relations that contain previously arrived primitive events (primitive event histories)* $oB_1, \ldots, oB_m$. Formally, $oB_i = \sigma\,[M_{Bi}<now](B_i)$

3. *Relations containing event histories of materialization points* $oQ_1, \ldots, oQ_n$. Formally: $oQ_i = \sigma\,[M_{Qi}<now](Q_i)$. Again, Q_i is the conceptual version of the materialization point that includes all events that have ever happened, in the past, present or the future.

After calculating the ΔQ_n, at the end of the evaluation step, $oB_1, \ldots, oB_m$ and $oQ_1, \ldots, oQ_n$ need to be updated (to $oB'_1, \ldots, oB'_m$, $oQ'_1, \ldots, oQ'_n$) so they can be used for the next evaluation step. The following two operations are performed at the end of the evaluation cycle to achieve this:

$oQ'_i = oQ_i \cup \Delta Q_i$

$$oB'_i = oB_i \cup \Delta B_i$$

Finally ΔQ_i and ΔB_i are flushed empty in preparation for a new evaluation cycle.

It is important to note that current results never rely on future events. If this were the case, then the event processing system would no longer be append-only.

6.2 Finite Differencing of TEQL Operators

TEQL extends and modifies the semantics of several CERA operators, as discussed in Chapter 5. Finite differencing equations need to be established for all operators that are used to formalized TEQL.

The following operators were added or modified in TEQL:

- The merge operator μ, was modified to allow modification of timestamp values and redefined as μ'
- The six relational operators UNION, INTERSECTION, DIFFERENCE, RESTRICT, PROJECT and JOIN were used for specification of temporal relational operators
- The PACK and UNPACK operators were added to the set of operators
- Project was allowed to drop non-timestamp interval attributes

To establish that finite differencing of TEQL is correct, it needs to be established that for all time points and all materialization points Q := E, where E is an extended CERA expression, for all R_i that are input relations to E and using the finite differencing equations for each operator, ΔE and oE are calculated correctly using ΔR_i and oR_i.

Due to temporal preservation this proposition can be stated based on the greatest end timestamp of each expression and input relation as (Eckert 2008):

If

$$\Delta R_i = \sigma[M_{Ri} = now](R_i) \text{ and } oR_i = \sigma[M_{Ri} < now](R_i)$$

then

$$\Delta E = \sigma[M_Q = now](Q) \text{ and } \mathrm{o}E = \sigma[M_Q < now](Q)$$

where M_E reflects the maximum end timestamp for a given expression E.

Essentially, this proposition requires that for each of the relational operators used in formalizing TEQL, the stated finite differencing equation is in fact correct for all points in time. Since the nodes of a query tree consist of the specified operator only, if the equation can be shown to be correct for each operator, it follows that any tree consisting of the operators will also be correct, due to structural induction.

The following sub-sections present the finite differencing equations for the extended CERA operators and discuss the correctness of each presented equation.

In this chapter, we will only address operators that have been added or modified in CERA. The interested reader is referred to (Eckert 2008) for the discussion of finite differencing for CERA operators in their original form.

6.2.1 PACK and UNPACK

The PACK and UNPACK operators are added to CERA to formalize temporal relational operators.

UNPACK

The following equations present the finite differencing of UNPACK:

$$\Delta(\mathrm{N}[interval\_name](\mathrm{E})) = \mathrm{N}[interval\_name](\Delta\mathrm{E})$$
$$\mathrm{o}(\mathrm{N}[interval\_name](\mathrm{E})) = \mathrm{N}[interval\_name](\mathrm{oE})$$

Since UNPACK can only be applied after a window join, we already know that M_E will be equal to the end timestamp of a window. Since UNPACK cannot be applied to the window boundaries, and does not cause an increase in the end time of the interval it is being applied to, it does not affect the value of M_E at any given point in time. Because UNPACK is a monotonic, non-blocking operator, the selection condition can be pushed through for both the Δ and o expressions:

σ [M_Q=*now*](N[*interval_name*](E)) = N[*interval_name*] (σ [M_E=*now*](E))
σ [M_Q<*now*](N[*interval_name*](E)) = N[*interval_name*] (σ [M_E<*now*](E))

It should be noted that for the restricted version of UNPACK, which can only be applied after a window join, the end timestamp of the window differentiates between current and previous results. However, the finite differencing formulas given above do not hold for the UNPACK operator, in general. As discussed in Section 5.4.1, UNPACK is a monotonic, non-blocking operator. This means that given an append-only input, the output of UNPACK will also be append only. However, since an unrestricted UNPACK would modify the value of the timestamp, the restriction conditions σ [M_Q=*now*] and σ [M_Q<*now*], cannot be used to calculate the Δ and o expressions.

PACK

The following equations present the finite differencing of PACK:

Δ(P[*interval_name*]E) = P[*interval_name*] ΔE
o(P[*interval_name*]E) = P[*interval_name*] oE

PACK is applied after a window join and the application of a relational operator. Since none of these operators are allowed to drop or modify the value of the window boundary, the value of M_E in the input of PACK is determined by the end time of the window boundary. In the output, since PACK cannot be applied to the window boundary and can not increase the value of the interval attribute it is being applied to, the value of M_E is still determined by the end time of the window boundary. Hence PACK cannot modify the value of M_E. Since PACK is applied after a window join, PACK is applied to tuples which fall in the window all at the same time. Tuples belonging to the same window fail or pass the σ [M_E=*now*] or σ [M_E<*now*] together. For each window, PACK can be calculated in an append only way, and the timestamp of the window differentiates between different windows. Hence, the select conditions can be pushed through:

$\sigma\,[M_Q{=}now](\mathrm{P}[interval\_name](\mathrm{E})) = \mathrm{P}[interval\_name]\,(\,\sigma\,[M_E{=}now](\mathrm{E}))$
$\sigma\,[M_Q{<}now](\,\mathrm{P}[interval\_name](\mathrm{E})) = \mathrm{P}[interval\_name]\,(\,\sigma\,[M_E{<}now](\mathrm{E}))$

As discussed in Section 5.4.1, an unrestricted PACK operator is blocking and non-monotonic. Clearly neither of the above equations are true for an unrestricted PACK operation.

6.2.2 MERGE

The operator μ' was defined in Section 5.5 to formalize the MERGE function as:

$$\mu'\,[\mathrm{DURING} \leftarrow i_1 \sqcup \ldots \sqcup i_n,\ b\_offset,\ e\_offset](E) \equiv \pi\,[\,\mathrm{DURING}.s \leftarrow (\min\{\,i_1.s, \ldots, i_n.s\,\}) + b\_offset,\ \mathrm{DURING}.e \leftarrow \max\{\,i_1.e, \ldots, i_n.e\,\} + e\_offset,\ sch(E)\,](E)$$

where $sch(E)$ is the set of attributes in the schema of E. The equations for finite differencing of μ' are given below:

$\Delta\,\mu'(\mathrm{E}) = \mu'(\Delta\mathrm{E})$
$o\,\mu'(\mathrm{E}) = \mu'(o\mathrm{E})$

If the offset values are given as zero, the μ' function becomes similar to the standard merge operator of CERA, μ, so we have:

$\Delta\,\mu'(\mathrm{E}) = \mu'(\Delta\mathrm{E})$
$o\,\mu'(\mathrm{E}) = \mu'(o\mathrm{E})$

If e_offset has a non-zero value, clearly the following equation does not hold.

$\sigma\,[M_Q{=}now]\,\mu'(\mathrm{E}) = \mu'(\,\sigma\,[M_E{=}now](\mathrm{E}))$

In fact we have:

$\sigma\,[M_Q{=}now{+}e\_offset]\,\mu'(\mathrm{E}) = \mu'(\,\sigma\,[M_E{=}now](\mathrm{E}))$

This is a case where the push through of the select condition does not hold true, but the equations for Δ and o can still be shown to be semantically correct. Semantically, Δ represents the new input and output tuples and o represents the past input and output tuples. μ' is essentially an extended projection which modifies the value of the timestamp

attribute. A project operation is monotonic, and given an append only relation as input, will also output an append only relation. Further, the output of μ' at each point now never causes a duplicate with previous outputs of μ'. This is due to the mandatory inclusion of *now+e_offset* in output for each step. Hence:

$$\Delta\, \mu'(E) = \mu'(\Delta E)$$
$$o\, \mu'(E) = \mu'(oE)$$

The following subsections cover the finite differencing of TEQL relational operators in the context of temporal relational operators. While some relational operators, are already covered in CERA, when they are used in the context of temporal relational operators, they are defined a specific way. Additionally, not all of these operators are supported in CERA, and need to be discussed here.

CERA already supports some of these operators in the standard context. These operators are discussed in Section 6.2.3. The finite differencing operators that are new to CERA are discussed in Section 6.2.4.

6.2.3 Pre-Existing Relational Operators

CERA already includes support for RESTRICT(SELECT), PROJECT and JOIN. This section discusses the finite differencing of these operators in TEQL making note of any differences these operators have with their original CERA specification, especially when used in the temporal relational operator context.

PROJECT

The definition of project is modified slightly from its original definition in CERA. In TEQL, project is allowed to drop non-timestamp interval attributes. Timestamp intervals cannot be dropped. Since only timestamp interval attributes are used in calculating M_E, this change does not affect the definition of the Δ and o operators for project. When used in the context of temporal project, a temporal project is not allowed to drop timestamp

intervals, hence it abides by the general restriction placed on project. The finite differencing equations for project remain unchanged:

$$\Delta\,\pi\,(E) = \pi\,(\Delta E)$$
$$o\,\pi\,(E) = \pi\,(oE)$$

These equations are not true for the standard project operator of relational algebra. A project operator is monotonic and hence, given an append-only input will produce an append-only output. Dropping of attributes could cause duplicates between current and previous results. Therefore $\Delta\,\pi\,(E) = \pi\,(\Delta E)$ is not true for the standard project operator.

RESTRICT

The definition of restrict has not been changed from CERA. TEQL does allow for specification of point-based conditions as a RESTRICT condition, which is not possible in XChange[EQ]. This does not affect the finite differencing of RESTRICT.

$$\Delta\ \sigma\ (E) = \sigma\ (\Delta E)$$
$$o\ \sigma\ (E) = \sigma\ (oE)$$

JOIN

The JOIN operator in CERA, does not have any special restrictions. The only difference between the JOIN used in TEQL and in standard CERA is that the TEQL JOIN is a θ-join, while the JOIN operation in CERA is a natural join. Both a standard join and a θ-join can be formulated as a restrict condition over a Cartesian product. Hence both versions of join are essentially monotonic, non-blocking operators and given append only inputs, will produce outputs according to the finite differencing formula:

$$\Delta(E1 \bowtie E2) = \Delta E1 \bowtie oE2 \cup\ \Delta E1 \bowtie \Delta E2 \cup oE1 \bowtie \Delta E2$$
$$o(E1 \bowtie E2) = oE1 \bowtie oE2$$

where $\bowtie$ is taken to represent both the natural and θ versions of join.

In the context of a TEQL temporal relational operator, JOIN is applied over windowed sources, with the following mandatory condition:

relation1_name.TEMP_DURING = *relation2_name*.TEMP_DURING

Since the temporal join occurs after the window join operator is applied, tuples participating in the temporal join will have the same value for the DURING attribute and hence the same value for $M_{E1}= M_{E2}$. This means that, in the case of a temporal join, the selection condition can, in fact, be simply pushed inwards as follows:

$$\sigma\,[M_Q{=}now](\,E1 \bowtie E2\,) = (\,\sigma\,[M_{E1}{=}now](E1) \bowtie \sigma\,[M_{E2}{=}now](E2)\,)$$
$$\sigma\,[M_Q{<}now](\,E1 \bowtie E2\,) = (\,\sigma\,[M_{E1}{<}now](E1) \bowtie \sigma\,[M_{E2}{<}now](E2)\,)$$

Hence the finite differencing equations for a join operation in the context of a temporal join can be simplified as:

$$\Delta(E1 \bowtie E2) = \Delta E1 \bowtie \Delta E2$$
$$o(E1 \bowtie E2) = oE1 \bowtie oE2$$

6.2.4 Operators Introduced to CERA

This section discusses the UNION, INTERSECT and DIFFERENCE operators.

UNION

CERA does not support an unqualified specification UNION operator. It is only supported as an auxiliary operator to combine different query results with each other. In TEQL, UNION is supported in the context of the temporal union operator. Finite differencing of a standard relational union is similar to a project: Union itself is non-blocking and monotonic. Given a pair of append only inputs, their UNION will also be append-only. However, an unqualified union operator may create duplicates with its history.

In TEQL, since UNION is used for specification of a Temporal Union, it is applied to the results of a window join. Hence at each point in time, tuples with the same M_E participate in the UNION simultaneously. This is why duplicates with history are not

possible, and M_E correctly differentiates between current and past results. The finite differencing of UNION can be specified as:

$$\Delta(E1 \cup E2) = \Delta E1 \cup \Delta E2$$
$$o(E1 \cup E2) = oE1 \cup oE2$$

INTERSECT

CERA does not include an INTERSECT operator. Support for INTERSECT is added in TEQL for specification of the temporal intersect operator. The standard relational intersect operator is monotonic, and non-blocking. Given a pair of append-only inputs, intersect returns an append-only result. In fact, the relational intersect operator can be seen as a special case of a natural join operator, where both relations share the same heading. Hence, relational intersect can be finite differenced similar to join as:

$$\Delta(E1 \cap E2) = (\Delta E1 \cap oE2) \cup (\Delta E1 \cap \Delta E2) \cup (oE1 \cap \Delta E2)$$
$$o(E1 \cap E2) = oE1 \cap oE2$$

Similar to a temporal join, since intersect is used in TEQL for specification of temporal intersection, it is applied over the output of a window join and hence can be simplified similarly as:

$$\Delta(E1 \cap E2) = \Delta E1 \cap \Delta E2$$
$$o(E1 \cap E2) = oE1 \cap oE2$$

DIFFERENCE

CERA does not include a difference operator. In TEQL, difference is used for specification of the temporal difference operator.

The unrestricted difference operator of relational algebra is a non-monotonic operator that, given a pair of append-only inputs, can cause removal of previously generated results at future points in time. However, in TEQL, difference is performed over the output of a window join. This makes the difference of each windowed operand an append-only operation. The output of one window does not affect previous results and

can be differentiated by the value of M_E, which is the end timestamp of the window. Therefore we have:

$$\sigma\,[M_Q{=}now](\,E1 \setminus E2)) = (\,\sigma\,[M_{E1}{=}now](E1) \setminus\ \sigma\,[M_{E2}{=}now](E2))$$
$$\sigma\,[M_Q{<}now](\,E1 \setminus E2)) = (\,\sigma\,[M_{E1}{<}now](E1) \setminus\ \sigma\,[M_{E2}{<}now](E2))$$

Consequently:

$$\Delta(E1 \setminus E2) = \Delta E1 \setminus \Delta E2$$
$$o(E1 \setminus E2) = oE1 \setminus oE2$$

6.3 Finite Differencing Example

All TEQL queries need to be finite differenced before they can be evaluated. This is done by first translating the query into the algebraic form and then calculating the Δ for the relational expression. The Δ and o operators are then pushed into the expression until no unqualified references to input relations remain in the query formula. In the final section of this chapter, the finite differencing of a complete temporal project statement is detailed for illustration purposes.

As presented in Chapter 5, a temporal project can be stated as:

TR=P[TEMP_DURING]
(π [*project_list*, TEMP_DURING, DURING]
(N [TEMP_DURING]
(W $\bowtie_{i \exists j}$ (ρ[TEMP_DURING/DURING](R)))))

The temporal project query has a single input event R. The temporal project is broken down into its constituent operators in top-down presentation below:

1. PACK:

   TR = P [TEMP_DURING] (R_4)

2. PROJECT:

   R_4 = π [*project_list*, TEMP_DURING, DURING] (R_3)

3. UNPACK:

   R_3 = N [TEMP_DURING] (R_2)

4. Window join:

$$R_2 = (W \bowtie_{i \sqsupseteq j} R_1)$$

5. Rename:

$$R_1 = \rho[\text{TEMP\_DURING/DURING}](R)$$

The goal is to arrive at the formula that can calculate ΔTR, based on ΔR and oR. Finite differencing of a temporal project is shown below:

Step1:

$$\Delta(P[\textit{interval\_name}]E) = P[\textit{interval\_name}]\ \Delta E \Rightarrow$$
$$\Delta R5 = P\ [\text{TEMP\_DURING}]\ (\Delta R4)$$

Step 2:

$$\Delta\ \pi\ (E) =\ \pi\ (\Delta E) \Rightarrow$$
$$\Delta R4 = \pi\ [\textit{project\_list}, \text{TEMP\_DURING, DURING}]\ (\Delta R3)$$

Step 3:

$$\Delta(N[\textit{interval\_name}](E)) = N[\textit{interval\_name}]\ (\Delta E) \Rightarrow$$
$$\Delta R3 = N\ [\text{TEMP\_DURING}]\ (\Delta R2)$$

Step 4:

$$\Delta(E1 \bowtie i \sqsupseteq j\ E2) = \Delta E1 \bowtie i \sqsupseteq j\ oE2 \cup \Delta E1 \bowtie i \sqsupseteq j\ \Delta E2 \Rightarrow$$
$$\Delta R2 = \Delta(W \bowtie i \sqsupseteq j\ R1) = \Delta W \bowtie i \sqsupseteq j\ oR1 \cup \Delta W \bowtie i \sqsupseteq j\ \Delta R1$$

Step 5 (using ΔR1 and oR1):

$$\Delta\ (\rho\ (E)) = \rho\ (\Delta E) \Rightarrow$$
$$\Delta R1 = \rho[\text{TEMP\_DURING/DURING}](\Delta R)$$

$$o\ (\rho\ (E)) = \rho\ (oE) \Rightarrow$$
$$oR1 = \rho[\text{TEMP\_DURING/DURING}](oR)$$

This completes the finite differencing of TR as no unqualified references to R remain in the query expression. As mentioned previously, the above procedure needs to be performed for any query in order for it to be evaluated in a step-wise manner by the system. The following chapter discusses the prototype implementation which serves as a proof of concepts for the TEQL query language features.

Chapter 7

PROTOTYPE IMPLEMENTATION

A prototype has been implemented to demonstrate the functionality supported by TEQL. This chapter presents the details of the prototype implementation. Section 7.1 presents an overview of the system. This implementation uses the SQL Server relational database management system for maintenance of the event histories and execution of relational query translations in each step of the evaluation cycle. Similar to other relational database management systems, SQL Server does not natively support the temporal interval type. The implementation of interval type and associated operators are addressed in Section 7.2. Section 7.3 discusses the steps that need to take place when a new event query specification is supplied by the user. As discussed previously, the evaluation of event queries occurs in a cyclic manner. Section 7.4 details the steps that occur within each evaluation cycle. Section 7.5 describes the method used for testing the prototype. Section 7.6 presents a summary of the chapter.

7.1 Overview of System Architecture

The prototype system architecture consists of three main components:

- An SQL database. To simplify the implementation, an SQL database is utilized for two purposes: 1) It serves as the storage unit for all primitive and complex events and 2) it serves as the execution engine for the query at each step of the evaluation cycle.
- Script Generator: The Script Generator takes the event query specification from the user and generates the SQL scripts for query evaluation and storage. The storage scripts are run against the SQL database to create the event stores. The evaluation scripts are placed in the query evaluation plan which is handled by the Query Engine.

- Query Engine: The Query Engine is in charge of receiving input events and executing the query evaluation cycle algorithm.

The main system components are illustrated in Figure 7.1.

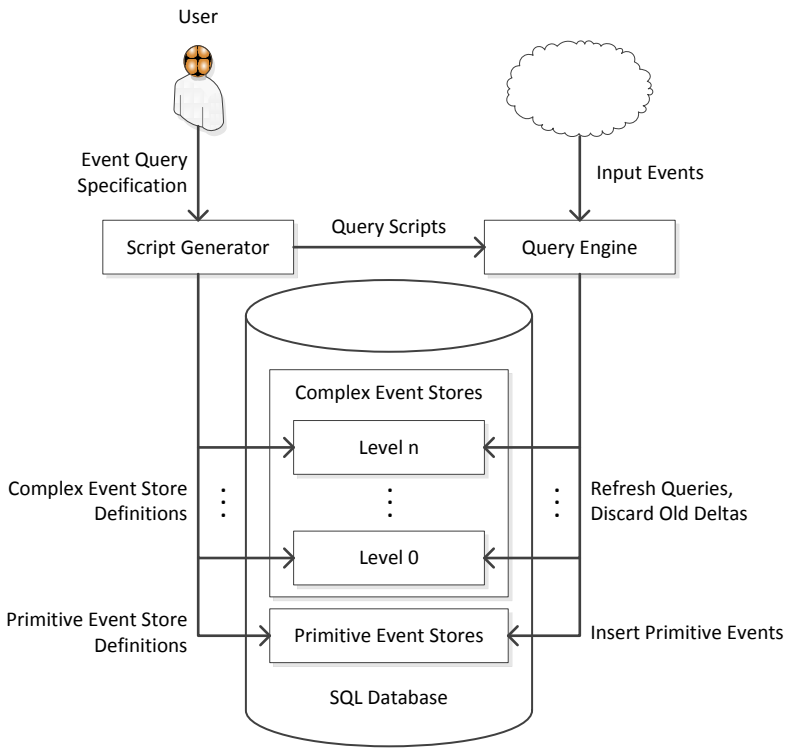

Figure 7.1. Overview of Prototype Architecture

Primitive events can be input from different sources. In the current implementation, a program has been implemented to simulate the linear road application scenario used for demonstrating TEQL functionality. The Query Engine receives the input events and places them in the correct primitive event history. The Query Scripts for complex events are organized into several levels. The complex events that make use of primitive events only are placed in level 0. Any complex events using primitive events and level 0 events are placed in level 1. In general, the level of each query is equal to the maximum level of its input events plus one. At each step of the evaluation cycle, the Query Engine evaluates queries in the ascending order of their levels. In this way, the result of each underlying query is ready when the higher level query requires it.

7.2 The Interval Type and Associated Operators

Support for the interval type and the PACK and UNPACK operators is not natively present in current mainstream RDBMSs. This is also true of Microsoft SQL Server. However, the importance of support for temporal databases seems to be gaining more attention. An indication of this can be found in the dedication of one chapter to the topic in the book (Ben-Gan, et al. 2009) by Microsoft Press. The implementation of the interval type and the PACK and UNPACK operators in this prototype are based on the solutions presented in this book. The following sub-sections briefly discuss the implementation of the interval type and the PACK and UNPACK operators based on (Ben-Gan, et al. 2009). The interested reader is referred to this reference for further discussion.

7.2.1 The Interval Type

SQL server has the ability to extend its base types with imported user-defined data types. These data types can be implemented in any .NET-based language, such as C#, and are called Common Language Runtime (CLR) User Defined Types (UDTs). (Ben-Gan, et al. 2009) introduces a CLR UDT called IntervalCID to represent the temporal interval type, where CID stands for Countably Infinite Discrete. Within the IntervalCID type, the begin and end time are represented by an integer value. The granularity of time is determined by a separate lookup table which maps the integer values to discrete points in time. Depending on the application, the lookup table can assume different units of time such as days, hours, seconds. The implementation of the IntervalCID type includes implementation of the following Boolean and set operators over intervals:

- Boolean (Allen's) Operators: Equals, Before, After, Includes, Properly_Includes, Meets, Overlaps, Merges, Begins and Ends
- Set Operators: Union, Intersect, Minus

Implementation details for the IntervalCID type and the Boolean and Set operators associated with it will not be presented here and can be found in (Ben-Gan, et al. 2009). Defining an interval type attribute in a table can be done as follows:

```
during    IntervalCID         NOT NULL,
```

Additionally, in order to properly view the begin and end value of the timestamp as integer values, the following two calculated attributes are added to the table definition:

```
beginint AS during.BeginInt PERSISTED,
endint AS during.EndInt PERSISTED,
```

The value of the interval type can be set as follows:

```
DECLARE @i1 IntervalCID;
SET @i1 = N'(4:8)';
```

where @i1 is a temporary attribute defined to demonstrate the assignment of a value to an IntervalCID type attribute. In the example above, the interval (4:8) is assigned as the interval value. The following subsections present the UNPACK and PACK operations.

7.2.2 UNPACK

A lookup table is used to represent the correspondence between the integer values used in the interval type and actual points in time. A join operation with the look up table can be used to generate unpacked tuples. The following code demonstrates the SQL code for unpacking a relation called WINDOWED_SOURCE:

```
SELECT
    ws.TEMP_DURING,
    N'(' + CAST(dn.n AS NVARCHAR(10)) +
    N':' + CAST(dn.n AS NVARCHAR(10)) + N')'
    AS unpacked_during
  FROM WINDOWED_SOURCE AS ws
    INNER JOIN DateNums AS dn
    ON dn.n BETWEEN ws.temp_beginint AND ws.temp_endint
```

where DateNums is the lookup table. In the WINDOWED_SOURCE relation, the interval is represented using the attributes named TEMP_DURING, temp_beginint and temp_endint. The join condition checks that the index of the lookup table falls between each begin and end

time value for the WINDOWED_SOURCE relation. The unpacked interval is stored in the attribute called unpacked_during.

7.2.3 PACK

The PACK operation over an unpacked input relies on the grouping operation as found in the standard SQL GROUP BY clause. The grouping factor requires some explanation. Essentially what is required is that all tuples having adjacent intervals to be placed in the same group. This grouping factor can be calculated based on the following two values:

- n: The index of the unpacked point in time in the interval, as found in the lookup table. When the join operation is performed with the lookup table each resulting tuple reflects a single point in time. For instance unpacking three tuples with the intervals (2:5), (5:6) and (8:9) will result in tuples reflecting 8 time-points with index values: 2, 3, 4, 5, 5, 6, 8, 9.
- The dense rank ordered by n. In SQL Server, the DENSE_RANK() function returns the rank of a row within a given order. For instance, for the given index values above, the DENSE_RANK() OVER (ORDER BY n) is 1,2,3,4,4,5,6,7 respectively.

Subtracting the value of n from the value of dense rank over n, produces the desired result:

n	rank order over n	n-(rank order over n)
2	1	1
3	2	1
4	3	1
5	4	1
5	4	1
6	5	1
8	6	2
9	7	2

The third column is the grouping factor for the PACK operation. The rank order grows in a linear fashion with the growth of n. Hence subtracting the value of n from the value of

rank order over n reflects any non linear growth in the value of n. Any non-continuous increase in the value of n is an indication of a new group of time points that need to be grouped together. Hence the PACK operation can be performed by performing a group by over the grouping factor defined above.

7.3 Script Generation

Full translation of a query in TEQL syntax into a query plan is left as future work. Using the prototype implementation, the user is able to specify a query by using functions that correspond to the language operators. By specifying appropriate parameters to the desired operators, the user is able to specify a desired query.

For a query to be evaluated, the following steps need to be performed:

- Corresponding event stores need to be created in the database for the delta and history of the event query.
- A query script corresponding to a finite differenced version of the requested operators needs to be generated.

These are one-time steps that need to be performed for any new query. Figure 7.2 illustrates these steps.

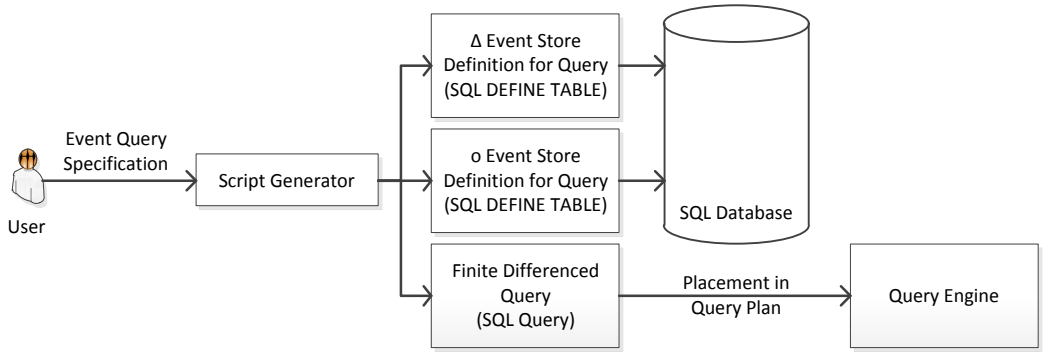

Figure 7.2 Query Script Generation and Execution

The role of the Script Generator is to create all three scripts. The SQL CREATE TABLE command is used for creating the delta and history event stores in the database. SQL queries are used to calculate the result of the finite differenced query at each step. The

following subsections present snippets of generated code from the highway example scenario to demonstrate the scripts used for event store creation and query execution.

7.3.1 Event Store Scripts

Event stores are database tables used for storing events that have arrived as primitive events, or were generated as query results defined as complex events. Each primitive and complex event requires two event stores corresponding to the delta and history of that specific event.

For instance, the segStr query, which returns events corresponding to the periods of time during which a vehicle spends time in each segment, uses the following SQL Table definition as the event history store:

```
CREATE TABLE ES.segStr
(
 vehicleid   INT         NOT NULL,
 segment   INT         NOT NULL,
 during      IntervalCID  NOT NULL,
 beginint AS DURING.BeginInt PERSISTED,
 endint   AS DURING.EndInt   PERSISTED,
 CONSTRAINT PK_segStr PRIMARY KEY(vehicleid, during)
);
```

where vehicleId is the vehicle identifier, segment is the segment number, and during is the duration that the vehicle spent in the segment. The interval attribute during is implemented using the IntervalCID type discussed in Section 7.2.1.

An identical table is created for the storage of the deltas:

```
CREATE TABLE ES.D_segStr
(
 vehicleid   INT         NOT NULL,
 segment   INT         NOT NULL,
 during      IntervalCID  NOT NULL,
 beginint AS DURING.BeginInt PERSISTED,
 endint   AS DURING.EndInt   PERSISTED,
 CONSTRAINT PK_D_segStr PRIMARY KEY(vehicleid, during)
);
```

The prefix D_ is used to distinguish the delta event store from the event history.

7.3.2 Finite Differenced Query

The finite differenced query is the query that is in fact evaluated at every cycle and calculates results associated with that cycle, which is the delta of the user specified event query. Finite differencing of TEQL operators was discussed in Section 6.2. Here we present the SQL script generated for the segStr query. Recall that segStr requires a self join of PosSpeedStr. Also recall that:

$$\Delta(E1 \bowtie E2) = \Delta E1 \bowtie oE2 \cup \Delta E1 \bowtie \Delta E2 \cup oE1 \bowtie \Delta E2$$

Since, in this specific example, E1 and E2 are the same event type, the finite differencing of JOIN can be simplified as:

$$\Delta(E1 \bowtie E1) = \Delta E1 \bowtie oE1 \cup \Delta E1 \bowtie \Delta E1$$

Hence, the SQL script for the finite differenced self join is generated as:

```
        SELECT
                P1.vehicleId, P1.segment, P1.dir, P1.hwy, P1.beginint, P2.endint
        FROM
                ES.PosSpeedStr as P1, ES.D_PosSpeedStr as P2
        WHERE
                P1.beginint < P2.beginint and
                P1.vehicleId = P2.vehicleId and
                P1.segment+1 = P2.segment
UNION
        SELECT
                P1.vehicleId, P1.segment, P1.dir, P1.hwy, P1.beginint, P2.endint
        FROM
                ES.D_PosSpeedStr as P1, ES.D_PosSpeedStr as P2
        WHERE
                P1.beginint<P2.beginint and
                P1.vehicleId = P2.vehicleId and
                P1.segment+1 = P2.segment
```

To generate the above script, the join conditions and input event stream names are provided as input by the user. The above script consists of the UNION of two identical join queries. One of the join queries is a self join of the delta with itself and the other is between the delta and the history. As can be seen, the primitive input event stream has the history and delta event stores already defined. The storage of the query results into the

relevant event stores is part of the evaluation cycle and is discussed in the next subsection.

When multiple steps are required to compose a query, the *Common Table Expression*, WITH statement, is used to name and reference each step. For instance, the temporal project operation requires the following operators:

- WINDOWED SOURCE
- UNPACK
- PROJECT
- PACK

In the highway example scenario, the script for the segBusy query which consists of a temporal project of segment over segStr is as follows:

```
WITH windowed_source AS (
	SELECT
		window.*, base.vehicleid, base.segment, base.beginint temp_beginint,
		base.endint temp_endint, base.during TEMP_DURING
	FROM
		ES.D_90_second_timer AS window, ES.D_segStr AS base
	WHERE
		window.beginint <= base.beginint AND window.endint >= base.endint
UNION
	SELECT
		window.*, base.vehicleid, base.segment, base.beginint temp_beginint,
		base.endint temp_endint, base.during TEMP_DURING
	FROM
		ES.D_90_second_timer AS window, ES.segStr AS base
	WHERE
		window.beginint <= base.beginint AND window.endint >= base.endint
),
unpacked_source AS (
SELECT
		ws.*, CAST(ws.TEMP_DURING.ToString() AS CHAR(8)) AS completeduring,
	dn.n,
	N'(' + CAST(dn.n AS NVARCHAR(10)) + N':' + CAST(dn.n AS NVARCHAR(10)) + N')'
	AS unpackedduring
FROM
	windowed_source AS ws INNER JOIN dbo.DateNums AS dn
		ON
			dn.n BETWEEN ws.temp_beginint AND ws.temp_endint
),
temporal_project AS (
SELECT DISTINCT
	segment, during, n, unpackedduring AS TEMP_DURING
FROM
	unpacked_source
),
GroupingFactorCTE AS (
SELECT
	segment, during, n,
	DENSE_RANK() OVER (ORDER BY n) AS dr,
	n - DENSE_RANK() OVER(ORDER BY n) AS gf
FROM
	temporal_project
)
SELECT
	segment, gf, during,
	N'(' + CAST(MIN(n) AS NVARCHAR(10)) +
	N':' + CAST(MAX(n) AS NVARCHAR(10)) + N')' AS packedduring
FROM
	GroupingFactorCTE
GROUP BY
	segment, during, gf
```

where the windowed source sub-query contains a finite differenced window-join. The PACK and UNPACK steps are applied as discussed in Section 7.2. The next section discusses the steps performed in each evaluation cycle.

7.4 The Evaluation Cycle

This section describes the query evaluation cycle that is executed by the query engine. Primitive events are received directly by the query engine. Complex events are organized into different levels based on their dependency on other events. Level 0 queries only depend in primitive events. Level 1 queries may depend on primitive events or level 0 queries, and so on. The operations performed in each step of the evaluation cycle are illustrated in Figure 7.3.

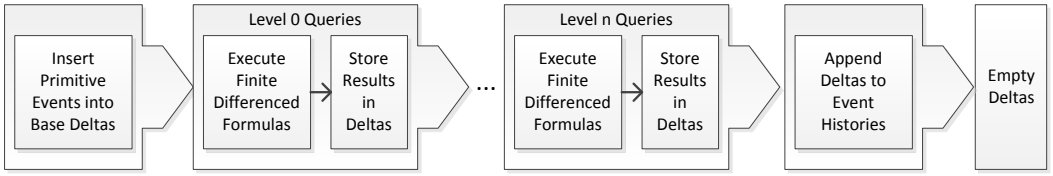

Figure 7.3 Steps in a Query Evaluation Cycle

The steps are as follows:

- The primitive events are received and inserted into the delta event stores already defined for each primitive event type.
- Finite differenced queries generated for complex events are executed and the results are stored in their appropriate delta event stores in the ascending order of their levels. At each level, the storage of query results in the delta event store occurs before moving up to the next level. In this way, the inputs for queries at each level are ready for the execution of the queries.
- After all query levels have been executed, for each primitive and complex event, the events residing in the delta event stores are appended to the corresponding history event store of each event.

- As a final step, and in preparation for a new evaluation cycle, the delta event stores are emptied at the end of each cycle.

While the system is running, the query evaluation cycle, is performed in a continuous loop. It is assumed that the user specifies queries either before the system starts or when the system is stopped. Also, since high performance is not a key goal of this research, it is assumed that the query engine has enough time to execute queries at each cycle before the new cycle starts.

7.5 Testing the Prototype

The linear road example scenario described in Chapter 4 has been used for testing the prototype implementation. Only the functionality of the prototype has been tested as the purpose of the prototype is to demonstrate the feasibility of concepts introduced in this dissertation. Investigation, testing, and improving performance related criteria is left as future work and is outlined in Chapter 9. Figure 7.4 illustrates the queries formulated in Chapter 4 and their dependencies.

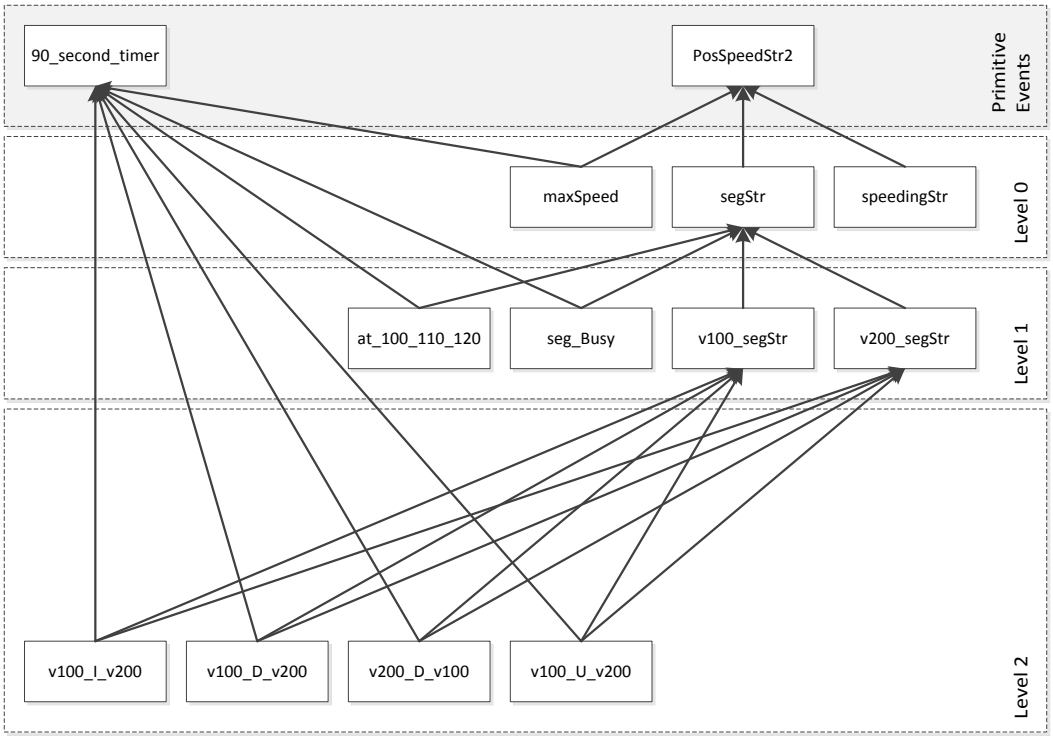

Figure 7.4 Dependency Diagram for the Linear Road Suite of Queries

There are two primitive events involved in the queries: PosSpeedStr2 and 90_second_timer. PosSpeedStr2 is generated using a simulator program which will be described shortly. Based on their dependencies, the queries can be organized into 3 distinct levels:

- Level 0: maxSpeed, segStr and speedingStr only use PosSpeedStr2 and 90_second_timer as input which are primitive events. Thus they are placed in Level 0.
- Level 1: at100_110_120, seg_Busy, v100_segStr and v200_segStr reference 90_second_timer and segStr. 90_second_timer is a primitive event, but since segStr is at level 0 these queries are placed in level 1.
- Level 2: v100_I_v200, v100_D_v200, v200_D_v100 and v100_U_v200 reference 90_second_timer, v100_segStr and v200_segStr. 90_second_timer is a

primitive event, but since v100_segStr and v200_segStr are level 1 queries, these queries are placed in level 2.

Stream simulator

A program is used to simulate the PosSpeedStr2 stream. The simulator program can generate data corresponding to multiple vehicles. The input to the simulator program is an array of vehicle parameters. The parameters supplied for each vehicle are:

- Id: The vehicle id.
- Start: The time at which the vehicle enters the highway, and starts generating data.
- Stop: The time at which the vehicle leaves the highway, and stops generating data.
- Speed: The movement speed for the vehicle.
- Startpos: The position at which the vehicle enters the highway.

At each cycle, if there are any vehicles at the end of a segment, the simulator generates the corresponding PosSpeedStr2 event. The function that calculates the position of each vehicle is as follows:

```
private void pos(int cycle)
{
   if (cycle < start || cycle > stop)
      position = -1;
   else
      position = startpos + speed * (cycle - start);
}
```

The position of each vehicle is calculated based on the point of time. The point in time is passed to the function using the parameter cycle. The calculated position for the vehicle is stored in the class level attribute position. If the vehicle has not entered the highway or has left the highway, a position of -1 is returned to indicate this. The unit for the position is one tenth of a mile.

The vehicle only generates a new PosSpeedStr event when it reaches the end of each segment hence the function that calculates whether a new event should be generated or not is as follows:

```
public int getSegment(int cycle)
{
   pos(cycle);
   if (position == -1)
      return -1;
   else
   {
      if ((position % 10) == 0)
         return position / 10;
      else return -2;
   }
}
```

The getSegment function calls the pos function. getSegment returns -1 if the car hasn't entered the highway or has already left the highway. getSegment returns -2 if the car isn't at the end of the segment. In this case, no PosSpeedStr is generated for this vehicle. If the vehicle is at the end of the segment, the segment number is returned. In this case, a PosSpeedStr event is generated for this vehicle, and the segment number for the vehicle is reported.

Correctness

The SQL scripts generated for the event queries have a one-to-one correspondence with the finite differenced formulae developed in Chapter 7. Hence the arguments made for the correctness of the operator formulas in Chapter 7 also hold for the equivalent SQL scripts.

7.6 Summary and Outlook

The prototype implementation leverages the existing database framework of MS SQL Server for storage of events and evaluation of relational queries at each step of the evaluation cycle. Even though simplifying assumptions have been made, the system enables the designation of any sub-query as a materialized result. Hence it builds a

platform for exploration of the choice of different materialization points on query performance. Pursuing this topic is beyond the scope of this research. The developed prototype facilitates future work in this area.

Chapter 8

EVALUATION OF TEQL EXPRESSIVITY

Every temporal relational operator introduced in TEQL enables a new kind of query to be specified over event streams. This chapter compares the use of temporal relational operators in an event setting against operators found in previous languages, focusing specifically on composition operator-based languages and operators based on Allen's temporal relationships as two points of reference. Example queries are presented as a basis to discuss the unique expressivity enabled through the use of temporal relational operators.

The comparisons are broken across the four different qualities observed in temporal relational operators as supported in TEQL. Section 8.1 describes the *Semantic Aggregation* property as observed in the temporal PROJECT and UNION operators. Section 8.2 presents the *Unit Based Interval-Output* property as observed in the temporal RESTRICT operator. Section 8.3 presents the *Interval Set Operators*, focusing on temporal INTERSECT and DIFFERENCE. Section 8.4 discusses the *Temporal Co-occurrence of Events* in the context of the Temporal Join operator. Section 8.5 revisits previous approaches and discusses some of the features not present in TEQL. Finally Section 8.6 presents a summary of the chapter.

8.1 Semantic Aggregation

Aggregate operations are useful for gathering meaningful summaries of data. Due to the high volume of data possible in event processing systems, reducing the volume of events is an important concern in this area. As a means for succinct communication of information, aggregation operations have extra significance in event processing systems. To this effect, different event and stream processing languages have sought to present various aggregation operators over size or count based windows. Many event languages

emulate the standard aggregation operators found in the database area, such as SUM, AVG or MAX. Besides using a simple sequence of events, delimited by a fixed count or period of time, some event and stream processing languages have also attempted to enable more expressive ways of grouping data together by introducing concepts such as *partitioned* (A. Arasu, Babu, and Widom 2006) or *semantic* (Qingchun Jiang, Adaikkalavan, and Chakravarthy 2007) windows. These language features were discussed in Chapter 2.

In an event processing environment, the temporal dimension is of centric importance. A logically natural and powerful candidate for merging data together is temporally adjacent data. The PACK operator is unique in its capability to provide aggregation of temporally adjacent events and has thus far not been supported in event stream processing languages.

While the definition of all temporal relational operators includes the PACK operation, the PROJECT and UNION temporal operators best reflect the semantic aggregation property of PACK due to their exclusion of UNPACK. Temporal UNION and temporal PROJECT have similar aggregation properties:

- Given a single input event stream, temporal PROJECT aggregates adjacent events having compatible data into a single event, disregarding the values for excluded attributes.
- Given multiple input event streams, with the same heading, temporal UNION aggregates adjacent events having compatible data into a single event, disregarding the source of the event.

Consider a smart office scenario with an input event that reports the periods of time during which each employee spent in a department. Consider the following query:

Return continuous periods of time during which at least one employee was present in any department.

It is easy to imagine similar queries in different environments. Essentially, this type of query requires an aggregation of the input stream of events, as supported in the PACK operator. In practice, in different scenarios, there might be different ways in which the input streams of events are made available. For instance, the following example scenarios may exist:

- A single input event stream called employee_dept can be used to report the location of different employees in different departments:

  Employee_dept(EID, DEPT,DURING)

  where EID is the Employee ID, DEPT is the name of the department, and DURING reflects the period of time during which the employee was present in the department.
- An alternative design might involve sensors worn by different employees, each reporting the period during which an employee was present in a specific department:

  Employee_1000(DEPT, DURING)
  Employee_2000(DEPT, DURING)
  …

Depending on how the input event streams are designed, the UNION or PROJECT operator can be used to perform the semantic aggregation:

- In the first case, a temporal PROJECT of DEPT and DURING, merges the temporally adjacent events pertaining to different employees which were present in the same department together

- In the second case, a temporal UNION of the input event streams, merges temporally adjacent events belonging to different streams, reflecting an employee being present in the same department together.

The following subsections take a more detailed look at the expressivity of temporal PROJECT and temporal UNION.

8.1.1 Temporal PROJECT

Temporal PROJECT has the ability to compose a new event based on data compatibility and temporal adjacency. This semantic and temporal aggregation reduces data which is of paramount importance in event environments. In this section, the new expressivity supported by temporal PROJECT is illustrated through comparison with a simple PROJECT operation supported in previous event languages.

Recall the following query from Section 8.1:

Return continuous periods of time during which at least one employee was present in any department.

With the following set of sample input data:

Employee_Dept

id	Dept	DURING
e1	CSE	[1001:1003]
e2	CSE	[1002:1003]
e3	ECE	[1006:1008]

a projection of Dept and DURING as supported in previous event languages would produce the following output:

regProject_Dept

Dept	DURING
CSE	[1001:1003]
CSE	[1002:1003]
ECE	[1006:1008]

As can be seen, a regular project operation, besides the reduced size of the heading, has no benefits in terms of reducing the volume of the input stream. If the implementation of PROJECT is based on the relational model, duplicates would be eliminated. But this would require all attributes, including the timestamp to have exactly the same values. Hence the expected reduction in the cardinality of the output events is minimal. Any overlap, or inclusion of one event by another would simply be missed by a regular project operation. In contrast, temporal PROJECT merges adjacent intervals with compatible data into a single event:

TempProject_Dept

Dept	DURING
CSE	[1001:1003]
ECE	[1006:1008]

With a temporal PROJECT, compared to a regular PROJECT, generation of redundant notifications of events corresponding to intervals that overlap or subsume each other is avoided.

8.1.2 Temporal UNION

Traditional composition-operator-based languages offer a disjunction operator, called OR. The function of the OR operator is to return a new event whenever either of its input events occur. Strictly speaking, the traditional OR operator cannot actually be categorized as a composition operator, as only one of the inputs plays a role for any output event that is generated. The output timestamp of the event is always equal to the event that triggered the OR operator.

In a smart-office setting, assume that employees have different roles, and we have two different derived streams which reflect the amount of time during which employees with the roles “manager” and “section manager” were present in a department. Both event streams have the same heading as shown below:

manager_dept(DEPT, DURING)

section_manager_dept(DEPT, DURING)

Now consider the following query:

Return continuous periods during which a 'manager' or a 'section manger' was present in a department.

Performing a disjunction of the two input streams would simply return all the events from both event streams. For instance if a section manager was present in a department during the period [10:20] and three different managers visited the same department for the durations [10:12], [14:16] and [19:22], the output of the disjunction would return all four events. The disjunction operator ignores the fact that these event intervals contain or overlap each other.

The temporal UNION operator is similar to a disjunction operator, in that occurrence of either inputs will cause the generation of an output. However the temporal UNION operator also aggregates temporally adjacent events into one event. For the above given intervals, the temporal UNION would result in one output event with the interval [10:22] being produced in the output. In this case, the UNION operator in fact creates new data, by reflecting the extended period during which events with similar data values were continuously true.

In summary, both temporal UNION and PROJECT have the ability to disregard a certain distinguishing factor between input events, and combine temporally adjacent events together. A temporal PROJECT is useful when the distinguishing factor to be ignored is the value of an attribute (or multiple attributes) in a single event source. A temporal UNION is useful when the distinguishing factor to be ignored is the source of the event itself. Using the semantic aggregation property supported by both of these operators results in the ability to communicate information by using fewer number of events and hence results in the important result of reducing total event volume. None of

the windowing features supported in previous event languages have the ability to specify temporal adjacency as an aggregation criteria.

8.2 Unit-Based Interval Output with Temporal RESTRICT

With the exception of Temporal PROJECT and Temporal UNION, all temporal relational operators involve unpacking of the input(s). This gives these operators the powerful ability to express conditions over intervals which are smaller than the input intervals. Likewise, the output of temporal relational operators can also report temporal data values which are smaller than the input events, possibly as small as the time unit of any given application. The reporting of interval data has not been supported in previous languages.

Temporal RESTRICT is particularly useful in illustrating the expressivity of temporal relational operators as it clearly demonstrates the ability to describe conditions on a sub-event-interval-length. Languages such as XChange^{EQ} which do not support UNPACK treat the interval of an event as an indivisible value. In these languages, intervals may be combined with each other to form a greater interval. But the new intervals can never be based on a portion of an input interval.

In TEQL, the application-specific unit of time is the basis for expressing queries and output generation. As a result of supporting UNPACK, the query conditions can be applied to sub-intervals as small as the application specific time granularity. Recall the Employee_Dept example from Section 3.2. Now consider the following query:

Return Employee number, Department Names, and continuous periods of time that the employees were present in the department later than [1002:1002].

with the following sample data:

Employee_Dept

id	Dept	DURING
e1	CSE	[1001:1003]
e2	CSE	[1002:1003]
e3	ECE	[1006:1008]

Answering the query correctly using previous event languages is problematic. First, even though the condition requires the presence of the employee in a department *after* a given point time, expressing this condition using operators based on Allen's relationships must be written as a *containment* condition. In other words the time point [1002:1002] needs to be contained in the period of time that an employee was present in the department. In general, point-based conditions can be emulated through a disjunction of Allen's relationships. As discussed in Section 4.5, this approach is cumbersome and prone to errors.

Aside from the correct capturing of the condition, a more significant issue with previous languages is the inability to produce the correct interval in the output. The correct output using the temporal RESTRICT operation needs to split the interval of the first tuple (e1, CSE, [1001:1003]) in half and report (e1, CSE, [1002:1003]) in the output. Using previous languages such as XChange^{EQ}, this is not possible. The condition might be correctly evaluated, but the interval in the output would be reported the same as it was in the input.

In summary, many of the previous languages are unable to capture temporal conditions that are supported using temporal RESTRICT. In newer languages such as XChange^{EQ}, that support Allen's operators, temporal equality and inequality conditions need to be translated into containment conditions, which is unintuitive. Even in these more expressive languages, the output event interval would not be accurate. These languages have the ability to check temporal conditions, but they have not been designed to report temporal data and simply report the length of the input events.

As demonstrated by the temporal RESTRICT operator, temporal relational operators are a powerful tool for querying of events as they allow derivation of temporal data regardless of the length of individual events.

8.3 Interval Set-Operators

UNION, INTERSECT and DIFFERENCE are known as set operators. Their use is best illustrated graphically using a Venn diagram. They are of paramount importance in set-theory to specify how members of different sets relate to each other. Intervals can also be viewed as sets of time points. In fact, sets of disjoint intervals are also sets of time points. Hence set operators are also a natural and powerful tool for describing the relationships between two sets of intervals. The diagram below illustrates two sets of intervals and their UNION, INTERSECT, and DIFFERENCE.

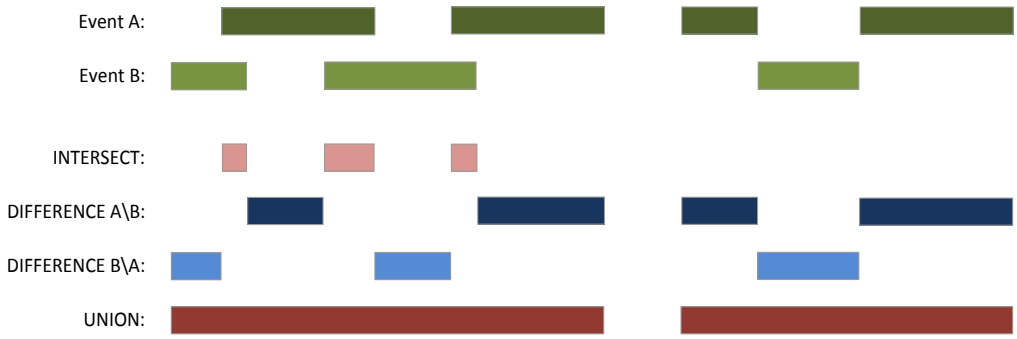

Figure 8.1. Temporal Patterns Captured Using Set Operators

Sets of intervals can be used to reflect the truth value of a proposition, as reflected in the data values of a tuple or an event. In this case the UNION, INTERSECT, and DIFFERENCE operators, respectively, reflect the periods of time where either of the propositions were true, both were true, or one was true but not the other.

While operators that seem to reflect similar relationships have been proposed in previous event languages, these operators do not correctly capture the temporal aspect of the relationships, as will be demonstrated shortly.

In short, when used as a temporal operator in an event setting, each set operator enables generation of events reflecting points in time where the specified set relationship was true between the incoming events streams. The expressivity of the temporal UNION

operators was already discussed in Section 8.1.2. The following subsections discuss the expressivity of Temporal INTERSECT and DIFFERENCE.

8.3.1 Temporal INTERSECT

An intersection of two sets returns items that belong to both sets. A temporal intersect operator applied to two event streams returns continuous periods of time during which both events were occurring and reflected the same data.

In previous event processing languages, the operator which is used to signify the co-occurrence of two events is the conjunction operator, AND. In most implementations AND simply pairs any two events from the specified sources together. Simply put, the conjunction operator is not concerned with when two events occurred with respect to each other. Rather it simply checks whether they have both occurred or not. In the point-based implementation of AND, the resulting event assumes the timestamp of the later occurring event. In the interval-based implementations of AND (Adaikkalavan and Chakravarthy 2005; François Bry and Eckert 2007), the output takes the begin timestamp of the earlier event and the end timestamp of the later occurring event. Neither the point-based, nor the interval-based implementations of the conjunction operator do in any way reflect the period during which the occurrence for both input events was true.

Consider the following query:

Return continuous periods of time during which both a manager and a section manager were present in the same department

The AND operator, coupled with a data constraint that requires the manager and the section-manager be in the same department, pairs all occurrences of manager_dept and section_manager_dept without considering the condition that they need to be in the department at the same time. Using XChange^{EQ}, it would be possible to formulate a temporal condition using a disjunction of Allen's operations, which would cover all the

different cases that the two intervals overlap each other. However the output interval would still follow the basic merge operator semantics as discussed in Chapter 6. In either case, the temporal period during which the two events actually ‘co-occurred’ would be lost.

In contrast, the temporal intersection operator returns exactly the period during which both events were true and shared the same data. Demonstrating the usefulness of the temporal intersection operator is easily possible with more example scenarios. For instance, a composite event that needs confirmation from different sources can be modeled based on the temporal INTERSECT operator. An example of this is a network intrusion detection event that relies on different event sources that report anomalous behavior independently. A temporal intersect would report the exact periods of time during which the input event streams concurred on the possibility of an intrusion, therefore reducing the chance of a false positive.

8.3.2 Temporal DIFFERENCE

The last set operator that is examined here is the difference operator. The difference operator, returns items that exist in one input set but not the other. The temporal DIFFERENCE operator returns event periods during which the first operand event occurred but a comparable event did not occur for the second operand.

Previous event languages have supported negation in the form of a non-occurrence event NOT. It has been observed that NOT cannot be considered an event operator in the composition sense, since no parameters from the subject event participate in a NOT operator. In this sense, it would be more accurate to view NOT as a condition, rather than an operator. Regardless of whether it is seen as an operator or a condition, NOT requires temporal boundaries over which to check for the non-occurrence of an event. When NOT is treated as an event operator, in a point-based setting, the end time of

the checking period is assigned as the occurrence time of NOT. Similarly, in an interval-based setting, the window boundaries over which the non-occurrence is checked is assigned as the window boundaries of the negation operator. In any case, NOT is not able to subtract the periods of time during which two different events were occurring from each other. This is a unique capability offered only by the temporal DIFFERENCE operator.

Returning to the network example, consider an event designed to detect a Denial of Service (DoS) attack. A DoS attack is associated with unusually high volume of network traffic. However there might be other mitigating factors that cause an increase in network traffic. For instance, a diagnostic test might also cause an increase in traffic. In this case, a temporal negation operator can be used to filter out the diagnostics cases from suspect network traffic. Consider the following input events stream: Diag_Test(IP, DURING) which reflects an ongoing diagnostics test for the given IP and High_Volume(IP, DURING) which reflects the detection of high volume data for a given IP address. The following query can be used for detection of DoS attacks:

> *Return continuous periods of time during which an IP address was experiencing a high volume of traffic but there was no diagnostics test for that address.*

Using the negation operator of previous languages will not yield the required results. For instance with XChange^{EQ}, even though a disjunction of Allen's relationships can be used to describe the temporal relationship required between Diag_Test and High_Volume, there is no way to output the intervals during which the occurrence of Diag_Test was true and High_Volume was not. In contrast, the temporal DIFFERENCE operator returns the exact periods of time as requested in the query. Temporal DIFFERENCE is useful in many queries where the interval of one event needs to be subtracted from the interval of another event. Another example scenario is in medical sensors where it is desirable to

create an alert for periods of time during which a patient experiences high blood pressure. However, there could also be benign causes for an increase in blood pressure, such as administration of a certain drug. Again the natural choice for expressing these kinds of queries is the temporal DIFFERENCE operator of TEQL rather than the negation operator found in previous event languages.

In summary, the NOT operator checks for the non occurrence of an event during a period of time, while temporal DIFFERENCE returns periods during which the occurrence of one event was true but not the other event.

8.4 Temporal Co-Occurrence of Events with Temporal JOIN

Temporal UNION, INTERSECT, DIFFERENCE and JOIN all demonstrate different ways to consider temporal co-occurrence of events. Temporal JOIN is unique in two regards. It allows the input events to have differing headings and it also allows direct specification of the JOIN condition. A discussion of the expressivity of Temporal JOIN follows.

The Temporal JOIN operator can be seen as a generalized Temporal INTERSECT operator where the input events are not required to share the same heading, and the qualifying condition can be expressed over a subset of the event heading attributes.

Similar to a temporal INTERSECT operator, the temporal JOIN operator can be compared with the conjunction operator, AND, found in other event processing languages. As mentioned previously, the AND operator is taken as the natural representation of the co-occurrence of two events. However, it does not reflect or even check for the temporal co-occurrence of the underlying events.

The temporal JOIN operator allows for expressing a JOIN condition over unpacked event data and is a powerful tool for expressing co-occurrence conditions. The

temporal JOIN operator offers a more accurate conception of co-occurrence of events as it takes into account when the input events were both actually occurring. Recall the smart office scenario from Chapter 4 with the following derived input event streams:

- empty_room(room_id, dept, DURING): an event reflecting the periods of time during which a given room was empty.
- on_light (room_id, DURING): an event reflecting the periods of time during which the light was on in a given room.

and the following query:

> *Return events, reflecting the room_id, department names and continuous periods of time during which a room was empty and the light was on.*

Performing the conjunction operator, AND, coupled with data filters to check for the sameness of the room_id does not take into account the requirement that the intervals for the two input events should share a non-zero period of time. In XChange^{EQ}, this temporal condition can be modeled through a disjunction of several of Allen's operators. However, the output event is still unable to capture the period of time during which the two events were both occurring. The correct result can be simply obtained by performing a temporal JOIN of the two input event streams.

Compared to a temporal intersect operator, a temporal join requires only the equality of attributes specified in the join condition. Similar to a temporal intersect operator, for each event generated, the temporal join operator returns the exact period of time during which the JOIN condition was true. A temporal INTERSECT operator can be seen as a special case of temporal JOIN where the input events share the same heading and the JOIN condition is equality of all attributes.

8.5 Unique Features of Previous Approaches

Section 3.4 presented a comparison of previous approaches to motivate incorporating temporal database features in an event language based on the relational framework of XChange^{EQ}. In this section the design of previous languages is revisited for comparison to TEQL. This section consists of three subsections which address the features of XChange^{EQ}, stream processing languages as exemplified by CQL and traditional composition based event processing languages.

8.5.1 XChange^{EQ}

Since TEQL is based on the same relational framework as XChange^{EQ}, TEQL supports most of the features present in XChange^{EQ}. The most notable exception to this is the fact that XChange^{EQ} is designed to query events which are provided in an XML format, while in TEQL events are akin to relational tuples. In this sense, TEQL is more comparable to Rel^{EQ}, which is a relational version of XChange^{EQ} used to demonstrate different language concepts. XChange^{EQ} integrates the Xcerpt language to support querying of XML data. In the relational specification, XChange^{EQ} uses two operators called matching and construction to map the XML data to tuples.

Many of the prominent projects in the area of event processing are focused on tuple-based events. As such it can be argued that querying tuple-based events is at least as important as querying XML events. Moreover tuple-based events are better aligned with the adopted approach for the TEQL design, which is strictly based on the relational model and temporal extensions to the relational model.

A second area of difference is that XChange^{EQ} supports triggering of actions through its support for reactive rules. In comparison TEQL, in its current form, only supports what are called deductive rules in XChange^{EQ}, which are only used for deriving new events.

8.5.2 CQL

CQL relies on the concept of temporal relations to support querying of streaming data. Temporal relations are relations that change over time and specify a limited window over the input streams. SQL-like queries can then be performed over the specified window. As discussed previously, the transformation of streams to relations and vice versa is performed through stream to relation and relation to stream operators. The windows themselves consist of count based, time based and partitioned windows.

In its current form, TEQL only allows specification of windows for non-monotonic operators. However, in theory, there are no obstacles to generalizing window specifications so that regular operators can be specified within windows also. Since in TEQL the window boundaries themselves are based on events, this results in a very powerful semantic window specification capability. But this also means that the type of windows that can be specified is dependent on whether the window specification itself can be expressed as a TEQL query. Simulating a count-based window using a COUNT() aggregate function would seem straightforward at first glance. However, since TEQL requires a separate window for the aggregate function specifications, this approach would not lead to an exact simulation of a count-based window. Since partitioned windows also rely on COUNT(), their exact simulation in TEQL is equally problematic. Simulating time-based windows is possible using the offset feature of the MERGE() function.

In general, the back and forth between streams and relations, as supported in CQL, leads to dual semantics and, at the very least, is not very intuitive. Additionally, the different window specifications are restrictive in terms of how and when the output stream can be generated. For instance, the RStream operator creates a stream with redundant results from multiple points in time. The DStream operator does produce a linear stream of output, however the query results can only be collected at the end of the window. For

instance, with a count-based window this means that the production of a new output from the window is dependent on when a new tuple arrives in the input. In the case of a time-based window, the output can only be generated after the passage of a specific amount of time, even if the result has already been calculated. In sum, the mandatory use of a window results in a lack of control over when and how outputs are generated. This is not a very versatile approach in applications where output event timings are important.

8.5.3 Traditional Event Languages

Traditional event languages rely on composition operators. A desirable consequence of this approach is the ability to chain multiple operators to express complex events succinctly. However, as discussed previously, the semantics of composition operators are mired with inconsistencies, and do not conform to expected equivalent formulations. Due to the numerous different relationships possible between interval-based events utilizing a composition operator-based approach in an interval-based setting does not seem intuitive. Furthermore, as discussed previously (Eckert 2008), composition operators mix two different querying dimensions: temporal conditions and event composition. This leads to limitations in expressivity.

TEQL already supports the different temporal relationships as specified by Allen (Allen 1983). The A-periodic operator found in earlier languages can be simulated through the use of a WINDWED_SOURCE operator. The window boundaries would be defined using an event query which detects the sequence of the two boundary events.

Some of the previous work in event processing has incorporated the concept of consumption modes. If implemented correctly consumption modes have the potential to limit the event output volume. However, many of the existing approaches mix event selection and consumption into a single dimension, which leads to limited options and confused semantics.

TEQL is based on the relational model. As such it pairs all input events against each other. This is sometimes called the comprehensive consumption in previous work. The selection of the first (or last) event among a set of events is theoretically possible in a relational setting, but would require a specification of a closed set of events and an ordering of those events. TEQL does support specification of a closed set of events through the WINDOWED_SOURC E operator. Selection of the last or first event within a window, while not currently supported, is straightforward.

Consumption modes modify the language behavior by specifying how and when underlying events will be removed from consideration. The relational model does not support discarding tuples through operator modifiers. While this might be theoretically possible, it seems that mixing arbitrary event consumption specifications within a relational language is not very intuitive. This would cause the language to behave in ways that are not expected in a relational language. This defeats one of the main attractions of TEQL: supporting consistently expected behavior mirroring the relational model. A more consistent approach would be to implement some form of garbage collection (Eckert 2008) to discard events that are no longer relevant. A garbage collection approach for discarding events does not affect the language behavior and only causes improved performance.

8.6 Summary

Temporal relational operators provide natural extensions to regular relational operators for their application to a temporal context. Application of these operators in an event setting, coupled with the open treatment of timestamps as supported in TEQL, leads to powerful new expressivity for reporting temporal data in event queries. This section presented four new dimensions of expressivity not present in previous event languages: Semantic Aggregation, Unit-Based Interval Output, Interval Set-Operators and Temporal

Co-occurrence of Events. TEQL does not support all features found in previous languages. The reasons for not supporting these features and the extent to which TEQL can support or simulate them was explained.

Chapter 9

SUMMARY AND FUTURE WORK

This dissertation has presented a new event query language, called the Temporal Event Query Language (TEQL). TEQL is based on the relational framework developed in (Eckert 2008) for evaluating relational statements in an environment. However the XChange^{EQ} relational framework is limited in its ability for querying temporal data. This dissertation extends the pre-existing framework to enable a more open treatment of temporal values and to enable temporal relational operators not previously supported in event languages. The development of TEQL includes formalization of language operators based on relational algebra. Efficient execution of relational statements over event streams requires a continuous, incremental evaluation which avoids re-computation of previous results. This is achieved through a technique called finite differencing. The finite differencing of different TEQL operators was discussed and formalized. A prototype has been developed to demonstrate the viability of the ideas presented in this dissertation. The prototype relies on an RDMS for the storage of events and evaluation of relational statements at each step of the evaluation cycle. The integrated approach of TEQL enables new kinds of expressivity for event queries based on well understood relational concepts. This chapter first presents a summary of the research contributions in Section 9.1 and then identifies key areas of future work in Section 9.2.

9.1 Summary of Research Contributions

Even though event languages have been a subject of study for a relatively long period of time, most of the work in this area has focused on a point-based conception of events. In recent years, semantic issues with the point-based model and business requirements have created a new focus on interval-based event languages. However, some of the problems

plaguing the point-based event languages, such as widely divergent event models and lack of a unified formal framework, have carried over to interval-based languages.

Temporal databases have also been studied for a few decades but have yet to gain commercial prevalence. Similar to event processing languages, temporal databases are concerned with storage and querying data that has a temporal dimension. Unlike previous event languages, querying of the temporal aspects of the data is the main focus of temporal database research. In contrast, previous event languages have focused on detecting temporal order of events instead of enabling querying timestamps as data values. This causes significant limitations on the kinds of event queries that can be expressed. An exception to this rule is the language XChange^{EQ}, which does separate the composition of events from declaration of temporal conditions. However, the work in (Date, Darwen, and Lorentzos 2002) in the temporal database area, which is the basis for the approach adopted in this dissertation, offers stronger expressivity over XChange^{EQ} in the following areas:

- Direct access to timestamps
- Support for temporal relational operators
- Support for multiple attributes with temporal data
- Control over output timestamp value

TEQL presents a new SQL-like event query language, which extends the XChange^{EQ} relational framework to support the expressivity benefits of the temporal database approach. Besides including Allen's operator for specification of temporal conditions, TEQL also allows for direct specification of conditions over timestamp values. This leads to easier specification of temporal conditions when time-points are being compared, rather than intervals.

TEQL includes support for the six temporal relational operators: RESTRICT, PROJECT, JOIN, UNION, INTERSECT and DIFFERENCE. These operators extend the corresponding 'standard' relational operators so that they are applicable to tuples tagged with interval values. They offer a powerful tool for deriving temporal information from events. This has not been possible with previous event languages.

The temporal relational operators are non-monotonic due to the inclusion of the PACK operation. Hence, they are required to be evaluated within a window when applied in an event setting. The WINDOWED_SOURCE operator is presented as a flexible way to specify windows over input events.

The MERGE() function allows the user to modify the output timestamp of a complex event by applying offsets to the default output timestamp event, which is based on the occurrence time of the input events.

In order to avoid any ambiguity, the operators in TEQL are formalized by extending the Composite Event Relational Algebra (CERA) (Eckert 2008). Standard relational algebra includes non-monotonic and blocking operators and hence is not appropriate for specification of event queries. CERA restricts relational algebra to avoid operator behaviors that are undesirable in an event setting. Collectively, these restrictions form the *temporal preservation* property of CERA. Extensions to CERA are necessary to support the new operators and features of TEQL such as the temporal relational operators or the MERGE() function. Upholding of temporal preservation has been demonstrated for the new operators and extensions to CERA.

An efficient execution of relational statements over event queries requires an incremental step-wise evaluation of the statements that avoids re-computing previously computed results at each step. The technique used to achieve this originates in the maintenance of materialized views and is called *finite differencing*. Finite differencing

entails differencing the input relations to two distinct relations: The current inputs (or the Δ) and past inputs (or o). The finite differencing for all new and extended CERA operators was developed and presented in this dissertation.

A proof-of-concept prototype has been developed to demonstrate the viability of the ideas presented in this dissertation. The prototype enables incremental step-wise evaluation of temporal event queries. The query results are materialized. Since the prototype supports using the output of other queries as input, any designated sub-query can also be materialized.

Finally, the expressivity offered by TEQL operators was compared directly against the closest comparable operators in previous languages. With its support for temporal relational operators, the kinds of new expressivity offered by TEQL can be summarized into the following main categories:

- Semantic Aggregation: The use of the PACK operator causes temporally adjacent events that are semantically compatible to be merged with each other. Given the issues with bandwidth limitation and processing of large data sets present in the event stream processing area, communicating the same semantic information with a minimum number of events is an important benefit of using the TEQL language. This property is best observed in the temporal PROJECT and temporal UNION operators.
- Unit Based Interval-Output: Because temporal relational operators (except PROJECT and UNION) operate over unpacked input intervals, they have the ability to specify conditions over and generate output events with intervals smaller than the input event intervals, possibly as small as a single unit of time within the given application. This property is best observed in the temporal RESTRICT operator. Previous event languages only compose whole events with

each other, do not support a non-timestamp interval attribute and force the resulting timestamp interval to encompass all constituent input event intervals.

- Interval Set Operators: UNION, INTERSECT and DIFFERENCE, are expressive operators that output intervals where one of the input intervals is valid, both are valid and one is valid but not the other.
- Finally the temporal JOIN operator is a powerful operator for checking co-occurrence of different event types. Similar to a RESTRICT operation, it allows for conditions to be specified over unpacked input events. However with join event streams with different headings can be considered against each other.

Besides the added expressivity, the TEQL language and its semantics are firmly based on the theoretical foundation of the relational model which is formally defined, prevalent and well understood. In contrast, many current languages do not include a proper formalization. When formalized, the approaches used for formalizing event processing utilize widely divergent models that are often not properly understood. Further, as seen in the related work chapter, many of the approaches that do offer some kind of formalization suffer from various issues in terms of ambiguity in semantics or expressivity.

Using the temporal relational model as a basis for event processing also allows for the exploitation of the large body of available work on optimization of query plans. Many of the current approaches utilize different, often unique, data structures for querying of events. As has been discussed before, using rigid data structures leaves no room for optimization based on reordering of operations.

Another benefit of using the temporal relational model is that, due to this common foundation with relational databases, extensions and development of relational database languages can easily be transferred over to the event processing realm.

The presented approach removes the mismatch between database data and event data. Since there is a direct correspondence between events and event types on one hand and tuples and relations on the other, querying data of both kinds simultaneously is straightforward. In many of the previous approaches, this correspondence is either non-existent or not immediately clear.

Finally, since the event data will eventually need to reside in a relational framework, it brings attention to the necessity of thinking ahead about the modeling of event data and mapping them correctly to a temporal relational framework. Issues such as identity and the relationships of different events to each other need to be sorted out in the initial design of the system, rather than left as an afterthought.

In summary this dissertation makes the following specific contributions:

- Development of a relational framework that enables processing of temporal event queries in an incremental manner,
- Development of TEQL, an SQL-like event query language with integrated support for temporal queries, including specification of semantics using the relational framework,
- Development of finite differencing formula for language operators, to enable step-wise evaluation of queries, and
- Implementation of a prototype of the language that demonstrates incremental evaluation of temporal event queries.

9.2 Future Work

As a new event query language the most important area of future work for TEQL involves optimized and efficient execution of event queries. As mentioned previously, a benefit of the developed framework is that it is based on a relational framework. As such, further research on TEQL offers ample opportunity for exploitation of previous work in

the areas of operator and query plan optimization. These areas have been studied deeply in the context of relational database management systems. This section identifies the key areas which can lead to enhanced performance for TEQL queries.

Choice of Materialization Points

The current framework supports materialization of different sub-queries. Materializing intermediate results makes it unnecessary to re-compute the sub-queries and hence results in faster query execution. However, more space is required for storage of intermediate results. Essentially, the choice of whether or not to materialize a sub-query involves a tradeoff between performance and storage which can be determined based on restrictions present in different situations. Additionally, since multiple queries may share a sub-expression, it is also possible to materialize the sub-expression once for use in all concerned queries. Studying the effect of the choice of materialization points on the performance of different operators under different event stream conditions is a future area of work.

Re-ordering of Query Operators

The developed framework mirrors a relational database management system modeling of a query plan. Similar to a database query, the order of operators in a query plan can have a significant effect on query performance. With databases, two of the criteria used for determining the order of operations in a query plan are the size of relations and the selectivity factor of the operands. In an event environment, since the events are arriving continuously, there is no notion of a fixed relation size. The volume of input events over time would be the equivalent criteria in an event setting. Also, it is important to note that in an event setting, the volume of events and the selectivity factors become dynamic criteria, which may change as a function of time. One area of future research is gathering

volume and selectivity factor on the fly and dynamically adjusting the query plan to adapt to input event characteristics.

Optimized Implementation of Temporal Operators

In the current implementation, the temporal relational operators are implemented based on physical PACK and UNPACK operations. Currently, the PACK and UNPACK operators are themselves implemented based on relational operators. It is possible, however, to implement temporal operators in a black box manner to execute more efficiently. In TEQL, PACK is applied at once, over closed sets of input events. However, if the input events are stored in an ordered manner, the PACK operator can be applied more efficiently. If A is the set of non-interval attributes in an event, and b is the begin time of the interval attribute, if the input events are ordered by the value of A, and then ordered by b among events that have the same A value, the PACK operation can be performed in a single pass (Date, Darwen, and Lorentzos 2002).

The UNPACK operator is an expensive operator in terms of space, however as indicated in (Date, Darwen, and Lorentzos 2002; Lorentzos and Mitsopoulos 1997) physical UNPACK can be avoided when only a single interval attribute is being unpacked, as is the case with TEQL queries. This requires direct implementation of the algorithms for each temporal operator. A future of area of work is the efficient implementation of temporal operators given the unique constraints placed on PACK and UNPACK in the TEQL framework.

REFERENCES

Abadi, Daniel J., Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. "Aurora: a New Model and Architecture for Data Stream Management." *The VLDB Journal The International Journal on Very Large Data Bases* 12 (2) (August): 120–139. doi:10.1007/s00778-003-0095-z.

Adaikkalavan, R., and S. Chakravarthy. 2005. "Formalization and Detection of Events Using Interval-based Semantics." In *Proceedings, International Conference on Management of Data*, 58–69. Citeseer.

Allen, J. F. 1983. "Maintaining Knowledge About Temporal Intervals." *Communications of the ACM* 26 (11): 832–843.

Arasu, A., S. Babu, and J. Widom. 2006. "The CQL Continuous Query Language: Semantic Foundations and Query Execution." *The VLDB Journal—The International Journal on Very Large Data Bases* 15 (2): 121–142.

Arasu, Arvind, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. "Linear Road: a Stream Data Management Benchmark." In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, 480–491. VLDB '04. Toronto, Canada: VLDB Endowment. http://portal.acm.org/citation.cfm?id=1316689.1316732.

Barga, Roger S, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. 2006. "Consistent Streaming Through Time: A Vision for Event Stream Processing." *Cs/0612115* (December 21). http://arxiv.org/abs/cs/0612115.

Barga, Roger S., and Hillary Caituiro-Monge. 2006. "Event Correlation and Pattern Detection in CEDR." In *Current Trends in Database Technology – EDBT 2006*, ed. Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, 4254:919–930. Berlin, Heidelberg: Springer Berlin Heidelberg. http://www.springerlink.com/content/tk7t177l27p58571/.

Ben-Gan,, Itzik, Dejan Sarka, Roger Wolter, Greg Low, Ed Katibah, and Isaac Kunen. 2009. *Inside Microsoft® SQL Server® 2008: T-SQL Programming*. 1st ed. Microsoft Press.

Bittner, Sven, and Annika Hinze. 2004. "Classification and Analysis of Distributed Event Filtering Algorithms." In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, ed. Robert Meersman and Zahir Tari, 3290:301–318. Berlin, Heidelberg: Springer Berlin Heidelberg. http://www.springerlink.com/content/g705rqv1e6p5ef8e/.

Brenna, Lars, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. 2007. "Cayuga: a High-performance Event Processing Engine." In *Proceedings of the 2007 ACM*

SIGMOD International Conference on Management of Data, 1100–1102. SIGMOD '07. Beijing, China: ACM. doi:10.1145/1247480.1247620.

Bry, François, and Michael Eckert. 2007. "Rule-based Composite Event Queries: The Language XChangeEQ and Its Semantics." In *Proceedings of the 1st International Conference on Web Reasoning and Rule Systems*, 16–30. RR'07. Innsbruck, Austria: Springer-Verlag. http://portal.acm.org/citation.cfm?id=1768725.1768728.

Bry, Francois, and Sebastian Schaffert. 2003. "The XML Query Language Xcerpt: Design Principles, Examples, and Semantics." In *Web, Web-Services, and Database Systems*, ed. Akmal B. Chaudhri, Mario Jeckle, Erhard Rahm, and Rainer Unland, 2593:295–310. Berlin, Heidelberg: Springer Berlin Heidelberg. http://www.springerlink.com/content/80wfu9v7r6uv84kg/.

Chakravarthy, S., and D. Mishra. 1994. "Snoop: An Expressive Event Specification Language for Active Databases." *Data & Knowledge Engineering* 14 (1): 1–26.

Cugola, Gianpaolo, and Alessandro Margara. 2010. "TESLA: a Formally Defined Event Specification Language." In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, 50–61. DEBS '10. Cambridge, United Kingdom: ACM. doi:10.1145/1827418.1827427.

Darwen, H., and C. J. Date. 2005. "An Overview and Analysis of Proposals Based on the TSQL2 Approach." http://www.dcs.warwick.ac.uk/~hugh/TTM/OnTSQL2.pdf.

Date, C. J., H. Darwen, and N. A. Lorentzos. 2002. *Temporal Data and the Relational Model*. Morgan Kauffman Publishers.

Date, C. J., and Hugh Darwen. 2000. *Foundation for Future Database Systems: The Third Manifesto*. 2nd ed. Addison-Wesley Professional.

Demers, A., J. Gehrke, M. Hong, M. Riedewald, and W. White. 2005. "A General Algebra and Implementation for Monitoring Event Streams."

Diao, Y., P. Fischer, M. J. Franklin, and R. To. 2002. "YFilter: Efficient and Scalable Filtering of XML Documents." In *Data Engineering, 2002. Proceedings. 18th International Conference On*, 341–342. IEEE.

Dong Zhu, and A. S Sethi. 2001. "SEL, a new event pattern specification language for event correlation." In *Tenth International Conference on Computer Communications and Networks, 2001. Proceedings*, 586–589. IEEE. doi:10.1109/ICCCN.2001.956327.

Eckert, M. 2008. "Complex Event Processing with XChangeEQ: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events". PhD Dissertation, Fakultat fur Mathematik, Informatik und Statistik: Ludwig-Maximilians-Universitat Munchen.

EsperTech, Inc. *Event Stream Intelligence: Esper & NEsper*. EsperTech Inc. http://esper.codehaus.org/.

Etzion, O., S. Jajodia, and S. Sripada. 1998. *Temporal Databases: Research and Practice*. Springer Verlag.

Galton, Antony, and Juan Carlos Augusto. 2002. "Two Approaches to Event Definition." In *Database and Expert Systems Applications*, ed. Abdelkader Hameurlain, Rosine Cicchetti, and Roland Traunmüller, 2453:547–556. Berlin, Heidelberg: Springer Berlin Heidelberg. http://www.springerlink.com/content/46mbb6ajt6t20qvd/.

Gatziu, Stella, Andreas Geppert, and Klaus R Dittrich. 1995. "The SAMOS Active DBMS Prototype." In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 480–. SIGMOD '95. San Jose, California, United States: ACM. doi:10.1145/223784.223893.

Gehani, N. H, H. V Jagadish, and O. Shmueli. 1992. "Event Specification in an Active Object-oriented Database." *SIGMOD Rec.* 21 (2) (June): 81–90. doi:10.1145/141484.130300.

Ghezzi, Carlo, Dino Mandrioli, and Angelo Morzenti. 1990. "TRIO: A Logic Language for Executable Specifications of Real-time Systems." *Journal of Systems and Software* 12 (2) (May): 107–123. doi:16/0164-1212(90)90074-V.

Gyllstrom, D., E. Wu, H. J. Chae, Y. Diao, P. Stahlberg, and G. Anderson. 2006. "Sase: Complex Event Processing over Streams." *Arxiv Preprint Cs/0612128*.

Hinze, Annika, and Agnès Voisard. 2002. *A Flexible Parameter-dependent Algebra for Event Notification Services*. Freie Universitat Berlin. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.494.

Lorentzos, N.A., and Y.G. Mitsopoulos. 1997. "SQL Extension for Interval Data." *IEEE Transactions on Knowledge and Data Engineering* 9 (3) (June): 480 –499. doi:10.1109/69.599935.

Luckham, D. C. 2007. "A Short History of Complex Event Processing. Part 1: Beginnings." *Online Only (http://complexevents. Com/wp-content/uploads/2008/02/1-a-short-history-of-cep-part-1. Pdf)*.

Mei, Y., and S. Madden. 2009. "Zstream: a Cost-based Query Processor for Adaptively Detecting Composite Events." In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, 193–206. ACM.

Morrell, John, and Vidich Stevan D. 2007. *Complex Event Processing with Coral8*. White Paper. http://www.coral8.com/system/files/assets/pdf/Complex_Event_Processing_with_Coral8.pdf.

Mühl, Gero, Ludger Fiege, and Peter Pietzuch. 2006. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc.

Patroumpas, K., and T. Sellis. 2006. “Window Specification over Data Streams.” *Current Trends in Database Technology–EDBT 2006*: 445–464.

Pietzuch, P. R, B. Shand, and J. Bacon. 2004. “Composite event detection as a generic middleware extension.” *IEEE Network* 18 (1) (February): 44– 55. doi:10.1109/MNET.2004.1265833.

Purich, P. 2010. “Oracle Complex Event Processing Getting Started 11g Release 1 (11.1. 1) E14476-03” (April).

Qingchun Jiang, R. Adaikkalavan, and S. Chakravarthy. 2007. “MavEStream: Synergistic Integration of Stream and Event Processing.” In *Digital Telecommunications, 2007. ICDT ’07. Second International Conference On*, 29. doi:10.1109/ICDT.2007.21. 10.1109/ICDT.2007.21.

Roncancio, Claudia L. 1999. “Toward Duration-Based, Constrained and Dynamic Event Types.” In *Active, Real-Time, and Temporal Database Systems*, ed. Sten F. Andler and Jörgen Hansson, 1553:176–193. Berlin, Heidelberg: Springer Berlin Heidelberg. http://www.springerlink.com/content/cuxlutwd3u4jy82n/.

Sánchez, César, Sriram Sankaranarayanan, Henny Sipma, Ting Zhang, David Dill, and Zohar Manna. 2003. “Event Correlation: Language and Semantics.” In *Embedded Software*, ed. Rajeev Alur and Insup Lee, 2855:323–339. Berlin, Heidelberg: Springer Berlin Heidelberg. http://www.springerlink.com/content/06eutrptglke0nw7/.

Snodgrass, Richard Thomas. 1995. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers.

Srivastava, Utkarsh, and Jennifer Widom. 2004. “Flexible Time Management in Data Stream Systems.” In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 263–274. PODS ’04. Paris, France: ACM. doi:10.1145/1055558.1055596.

Walzer, Karen, Tino Breddin, and Matthias Groch. 2008. “Relative Temporal Constraints in the Rete Algorithm for Complex Event Detection.” In *Proceedings of the Second International Conference on Distributed Event-based Systems*, 147–155. DEBS ’08. Rome, Italy: ACM. doi:10.1145/1385989.1386008.

White, W., M. Riedewald, J. Gehrke, and A. Demers. 2007. “What Is Next in Event Processing?” In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 263–272. ACM.

Wu, Eugene, Yanlei Diao, and Shariq Rizvi. 2006. “High-performance Complex Event Processing over Streams.” In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 407–418. SIGMOD ’06. Chicago, IL, USA: ACM. doi:10.1145/1142473.1142520.

Zaidi, A. K. 1999. “On temporal logic programming using Petri nets.” *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans* 29 (3) (May): 245–254. doi:10.1109/3468.759269.

Zimmer, D., and R. Unland. 1999. “On the Semantics of Complex Events in Active Database Management Systems.” In *Data Engineering, International Conference On*, 392. Los Alamitos, CA, USA: IEEE Computer Society. doi:http://doi.ieeecomputersociety.org/10.1109/ICDE.1999.754955.