Parallel Event Detection in Active Database Systems:
The Heart of the Matter

Ulrike Jaeger
Humboldt-Universität zu Berlin
Institut für Informatik
10099 Berlin
Germany
jaeger@dbis.informatik.hu-berlin.de

Johann K. Obermaier
ABB Corporate Research AG
CHCRC-C2
Segelhof, CH-5405 Baden
Switzerland
johann.obermaier@cher.cab.ch

Abstract. This paper proposes a strategy for parallel composite event
detection in active database management systems. Up to now, event
detection is sequential and totally synchronized, and thus preserves the
timely order of events during the detection process. However, in dis-
tributed and extensible applications events may occur simultaneously in
parallel unsynchronized streams. In order to adapt composite event de-
tection to these new requirements we relax the timely order of events to a
partial order and process parallel event streams. As a consequence, com-
posite event detection must deal with unsynchronized and parallel event
compositions. Our approach introduces a hybrid parallelization strategy
for composite event detection in Active Database Management Systems
that respects the timely order of events.

1 Introduction

This paper discusses basic approaches to parallel detection of event com-
positions. Although it characterizes several techniques by means of a simple cost
model, it emphasizes on the aspects of order preserving parallelism and does not
cover performance issues. Parallel detection of event compositions is a facet of
the SMILE approach [Jae95] which extends the concepts of active database man-
gagement systems (ADBMS) in order to meet the requirements of event driven,
long lived, distributed, and partially connected applications. Events might oc-
cur simultaneously and could be processed in parallel. We adapt techniques and
optimizations from query execution in relational database management systems.
However, queries and composite events differ fundamentally. The semantics of
query execution is set oriented, whereas composite event detection is based on
open streams of events. Cardinality and frequency of incoming events is unpre-
dictable. Event compositions often base on a order-sensitive combinations, e.g.
sequences. Parallel detection destroys the timely order of events and composi-
tions.

Up to now, ADBMS detect events and event compositions sequentially. The
existing execution models guarantee a timely ordered detection process, but
become inefficient if many events and event compositions are involved.
The remainder of this paper is organized as follows: Section 2 sketches the state of the art in ADBMS with an emphasis on composite event detection by operator graphs. Section 3 discusses parallelization techniques developed for query execution in relational DBMS. We illustrate the different techniques by examples for detecting composite events. We discuss the special problems that arise in composite event detection if we naively adapt parallel query execution strategies. As a solution, Section 4 introduces our hybrid parallelization strategy for composite event detection. Section 5 discusses related research. Section 6 contains our conclusion and future work.

2 State of the Art in Composite Event Detection

ADBMS extend the regular DBMS functionality by event-condition-action rules, called ECA rules [DBM88]. An event represents the successful execution of some operation within the database system or application. The condition tests the context state when the event occurs. The action is executed, when the condition evaluates to true. The application defines and produces atomic events, and requests for atomic as well as complex composite events. The event detector component in ADBMS collects and distributes atomic events, and detects event compositions.

2.1 Basic Concepts

Events and History. An event is a "happening of interest" [GJS92b] within the ADBMS or its application. Events occur repeatedly; therefore those events are instances of a given event type. Events are ordered by global time stamps. The ADBMS attaches a time stamp to each event. The time stamps are isomorphic to $\mathbb{R}$. In contrast to most ADBMS approaches, our approach also integrates simultaneous events.

We denote event types by capital letters. Instances of an event type are represented by tuples, having a set of attributes like type, time stamp and others. In this paper we focus on type and time information only, therefore denote an atomic event as an aggregate $<\text{typename}>,<\text{timestamp}>$. For example, $A.4$ is an instance of type $A$, bearing the time stamp 4. Throughout this paper we use capital letters for type names as well as for the set of instances of those types.

During runtime, the ADBMS receives atomic events and collects those in a timely ordered history. For example, a history of event instances of types $A, B, C, D$ is: $(A.1, B.4, C.5, D.5, D.7, C.8, A.9)$ We will refer to this history throughout this paper.

Event Compositions. Events issued by the application are atomic events. Composite events are constructed from atomic or other composite events according to the semantics of event operators. Composite events are typed as atomic events. A composite event type $X$ refers to a certain expression of constituent event
types — except \( X \), as recursion is not allowed — and operators. A composite event instance \( x \) is a typed representation for a combination of constituent event instances. Languages for event compositions provide a variety of operators ([BZBW95], [PW93], [WC94]). Without discussing those languages, we classify event operators into three groups:

- **constructors** combine events from different sources to form new result tuples.
- **collectors** collect events from different sources and merge them without constructing a new event tuple.
- **selectors** receive events from one source and select a certain subset without constructing a new event tuple.

In this paper, we focus on a common subset of event operators as examples for each operator class. Let \( A, B \) be event types:

- **BEFORE \((A,B)\)**: instances of \( A \) are composed with instances of \( B \) if \( a \in A \) happens before \( b \in B \), i.e. time stamp of \( a \) < time stamp of \( b \). The operator is a constructor for triples \((a,b,t) \in A \times B \times TIME\), where \( t \) is the time stamp of \( b \).
- **AND \((A,B)\)**: instances of both \( A \) and \( B \) are composed, no matter what timely order. The operator is a constructor for triples \((a,b,t) \in A \times B \times TIME\), where \( t \) is the time stamp of the most recent of the two constituent events.
- **OR \((A,B)\)**: instances of either \( A \) or \( B \), no matter what timely order. The operator is a collector of events from \( A \cup B \). The result of \( OR \) is a heterogeneous set of instances of both types \( A \) and \( B \).
- **FIRST\((A)\)**: the oldest instance of \( A \). The operator is a selector of events for a single \( a \in A \), with time stamp of \( a \) ≤ time stamp of \( \tilde{a} \), \( \forall \tilde{a} \in A \).
- **LAST\((A)\)**: the most recent instance of \( A \). The operator is a selector of events for a single \( a \in A \), with time stamp of \( a \) ≥ time stamp of \( \tilde{a} \), \( \forall \tilde{a} \in A \).

We denote composite event types by capital letters as well. Instances of a given composite event type have the list of constituent events and a time stamp as arguments. For example, \( Z.5(A.2, B.5, 5) \) is an instance of the composite event type \( Z \), based on the composition \( Z = AND (A,B) \). \( Z.5 \) inherits the time stamp 5 from the most recent constituent event \( B.5 \).

**Composition Semantics.** The detection operates on sets of constituent events and produces sets of compositions. **Sentinel** ([CKAK94], [Kri94]) was the first ADBMS to provide a set of explicit language concepts to define which subset of possible combinations is required. The semantics of event composition is described by so called consumption modes. The composition semantics determines the behavior of constructor operators, like **BEFORE** and **AND**. There are many variations of composition semantics (cf. [Beh95], [Kri94]). In this paper we refer to the most general consumption semantics, called **ALL**, and the most common consumption semantics in ADBMS, called **CHRONICLE**.
Constructors with \textit{ALL} semantics are based on the Cartesian product of constituent events. Combinations are constructed according to the operator semantics. For example, \textit{ALL} (\textit{AND} \((A, B)\)) results in a set \(S\) with: \(|S| = |A| \times |B|\) and \(S \subseteq A \times B \times \text{TIME}\).

Constructors with \textit{CHRONICLE} semantics produce a subset of the \textit{ALL} result; combination of events also means consumption. If an event is combined for a composite event, it is not reused for others. \textit{CHRONICLE} describes the stream based behavior of most of the event detection implementations in ADBMS. For example, input events for \textit{AND} are combined and consumed in first-in-first-out (FIFO) order.

An ADBMS provides composite event detectors that automatically collect and combine constituent events of the event composition. In this paper, we discuss the stream based operator graph approach as used in \textsc{sentinel} ([CKAK94], [Kri94]), \textsc{reach} [BBKZ92], \textsc{adl} [Beh95], \textsc{smile} [Jae95], and others ([PW93], [WC94]). Variations of the graph approach are: finite automata in \textsc{ode} ([GJS92b], [GJS92a]), and modified colored Petri nets in \textsc{samos} ([Gat95], [GD92], [GD94]).

\textit{Operator Graphs.} An operator graph is represented by a set of nodes and edges. An edge indicates a stream of totally ordered entries. Entries are appended at the end of the stream and received from the head. A node is either a leaf, a root, or an operator node. A detection graph forms a tree.

- A \textit{leaf node} is labelled by an event type. It represents the perception of constituent events of that type. It has one input stream for the required constituent event type and one output stream. \textsc{leaf}(\textit{<type>}) performs selections on input streams, thus it is a selector operator.
- An \textit{operator node} represents an operator of the event language. It receives entries from a set of input streams and composes new result entries according to the operator definition. Some operators need internal buffers. The result composition is sent to the output stream. The output stream is input stream for another node.
- A \textit{root node} has one input stream and a set of output streams. The root prepares each entry for output to consumers. The output of \textsc{root}(\textit{<type>}) is a stream of composition instances of type \textit{<type>}. It sends the result to the application and other consumers of the composition. The root is a selector operator.

\textit{Example.} Throughout this paper we refer to the detection tree for the composition \(X=\text{	extsc{or}(A, \text{before}(B, \text{and}(C, D)))}\) as shown in Figure 1.

\subsection{2.2 Composite Event Detection}

Throughout this paper we use a very simple cost model:

- Each operator performs its work on a stream entry within one time unit, called tick.
For illustration, we show the behavior of operator trees as tables. Vertically we show the counter for ticks. Horizontally we show the operators of the operator tree. A field in the table contains the result of an operator at a special tick. Leaf operators receive constituent events from a given history. Each entry is represented by the aggregate \(<\text{type}>,\langle \text{time stamp} \rangle\). Operator nodes receive and combine incoming event. The result is a composition represented as a tuple \((\langle \text{constituent events} \rangle, \langle \text{time stamp} \rangle)\). Example: Table 1 shows the sequential behavior of a operator section for \(\text{BEFORE} (A, B)\). At tick 1 \(\text{LEAF}(A)\) accepts \(A.14\), at tick 2 \(\text{LEAF}(B)\) accepts \(B.17\). At tick 3 the \(\text{BEFORE}\) operator produces the composition of \(A.14\) and \(B.17\), bearing the time stamp 17.

<table>
<thead>
<tr>
<th>ticks</th>
<th>LEAF(A)</th>
<th>LEAF(B)</th>
<th>BEFORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A.14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>B.17</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>((A.14, B.17, 17))</td>
</tr>
</tbody>
</table>

Table 1. Example table for \(\text{BEFORE}(A,B)\)

Composite event detection in ADBMS so far is sequential and centralized ([Beh95], [BBKZ92], [CKAK94], [Gat95], [GD92], [GJS92b], [Jae95], [Kri94], [PW93], [WC94]). For composite events the detection process advances step by step with each constituent event. The composite event detector accepts events
in a totally ordered stream. For the sequential synchronized execution of the operator graph, we assume that new constituent events are sent to the leaf nodes synchronously. For each input entry the operator tree performs all possible operators, and produces results possibly up to the root. New input is accepted by the tree only if no further construction of results is possible. This strategy synchronizes event composition and guarantees that no event can overtake previous events during the detection process. For the ALL semantics constructor operators like AND and BEFORE have to buffer their input streams and reuse them for new combinations. For the CHRONICLE semantics the operands are not buffered. Input streams are received, combined, and consumed.

Sequential Example. Table 2 shows the execution of the example detection tree of Figure 1 on the history \( \{ A,1,B,4,C,5,D,5,D,7,C,8,A,7 \} \) using the ALL semantics. For brevity we omit the root.

<table>
<thead>
<tr>
<th>Ticks</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>AND</th>
<th>BEFORE</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>B,4</td>
<td></td>
<td></td>
<td>C,5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>B,4</td>
<td></td>
<td>D,5</td>
<td></td>
<td>(C,5, D,5, 5)</td>
<td>(B,4, (C,5, D,5, 5), 5)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>B,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>B,4</td>
<td></td>
<td>D,7</td>
<td></td>
<td>(C,5, D,7, 7)</td>
<td>(B,4, (C,5, D,7, 7), 7)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>B,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>B,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>B,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>B,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>B,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>B,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>C,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>C,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>C,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>C,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>C,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>C,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>C,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>C,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>C,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Sequential example for \( \text{ALL}(\text{OR}(A, \text{BEFORE}(B, \text{AND}(C,D)))) \)

Sequential CHRONICLE Example. Table 3 shows the execution of the same detection tree and history using the CHRONICLE semantics. Here, \( B,4, C,5, \) and \( D,5 \) are consumed by combinations. We indicate the deletion from the operator’s
buffers by a star (*) at consumption time. Entries above the * are thus marked as consumed. For example, $B.4$ is consumed at composition time tick 8. With consumption, $C.8$ and $D.7$ cannot lead to new complete composite events as they would in the ALL semantics.

<table>
<thead>
<tr>
<th>Ticks</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
<th>AND</th>
<th>BEFORE</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A.1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>$B.4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>C.5</td>
<td></td>
<td>$D.5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$*$</td>
<td>$*$</td>
<td>$(C.5,D.5,5)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$*$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(B.4,(C.5,D.5,5),5)$</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>$D.7$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>C.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(C.8,D.7,8)$</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>A.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A.9</td>
</tr>
</tbody>
</table>

Table 3. Sequential example for $\text{CHRONICLE(}OR(A,\text{BEFORE}(B,\text{AND}(C,D))))\text{)}$

### 2.3 Problems of Sequential Event Composition

The examples shows the effects of synchronization:

- Synchronized sequential event detection is time order preserving. Each row shows one result, meaning each operator execution blocks other operators. In each column the results are timely ordered. The execution does not destroy the timely order, due to synchronization. For example, in Table 3 the event $A.9$ is accepted only after $C.8$ has caused results up to the $OR$ operator. $A.9$ cannot overtake predecessors which would corrupt the timely order of the result tuples.

- Synchronized sequential event detection is not efficient. Since only one operator is active at a time, the others are idle. The small example shows a tendency that can be generalized for real world applications where many events occur. The ALL semantics causes exponential increase of intermediate results within the tree. New events will be accepted with increasing delays.

We looked for ways to speed up the composite event detection. In long running applications the number of events and intermediate result explodes exponentially. The sequential execution leads to considerable delays in composite
event detection, especially for the ALL semantics. As our environment is distributed, we discuss the effects of possible parallel event detection.

3 Two Basic Paradigms for Parallel Event Detection

Parallelization is especially useful for operator trees as introduced in section 2. Stream based operators are well suited for parallel execution. We could relax the synchronization: while operator nodes compute results, the tree accepts new events and computes result sets in parallel to produce the complete set of final results much faster.

In general, operator trees can be executed in parallel by three different strategies: First, inter-tree-parallelism: each detector is executed as a single process, all detectors run in concurrent processes. Second, inter-operator-parallelism: each operator in a tree is executed in a single specialized process. Third, intra-operator-parallelism: each operator is executed by a set of concurrent processes, ideally one for each data entry. All three parallelization strategies can be combined. Inter-tree-parallelization alone leads to a heavy workload for each process, since a single process evaluates all data entries from a leaf to the root in a single sequential program. Both inter- and intra-operator-parallelism add to a much finer granularity and diminish the workload for each process such that we achieve a much better load balancing between processes. In this section, we investigate inter- and intra-operator parallelism and apply it to our language operators for examples. We call the first pipelining parallelism strategy (PPS), the latter universal parallelism strategy (UPS).

3.1 Pipelining Parallelism Strategy (PPS)

Each operator is performed by a specialized process. Operators are connected by sequential streams. We assume that events are sent to the leaves in partial timely order.

After a start-up delay all operators receive and produce entries concurrently. However, PPS has a serious load balancing problem: while the workload for each operator increases towards the root, the grade of parallelism decreases. The root is a bottleneck with heavy workload. For our operator categories execution works as follows:

- Selector operators like LEAF and ROOT receive a single input stream, select the appropriate entries and send them to the output stream. For example, the leaf operator for type A pipes all entries of type A to the output stream.
- Constructor operators like AND and BEFORE have buffers for each input stream. For each input entry $e$ of one operand the buffer of the other operand is searched for appropriate partners. For example, BEFORE searches the buffers for entries with appropriate time stamps, and combines pairs. For each entry there may be a set of matching pairs. That set is sent to the output stream; one at a time. Then the entry $e$ is added to the corresponding buffer.
Fig. 2. Pipelining Parallelism Strategy

- Collector operators like OR receive entries from their input streams and pipe them to the output stream.

Time-sensitive operators like FIRST and LAST are special selectors. If the input stream is partially ordered by time stamps, the top element of the input is the first event, and the bottom element — if the stream is finite — will be the last event. Thus, FIRST produces the result very quickly and then ignores all subsequent entries, while LAST blocks the tree until the very last entry arrives. The LAST operator is undefined if the input stream is infinite.

Example. We use the example in for ALL (OR (A,BEFORE (B,AND(C,D))))). The events enter the tree in parallel. We send events in a partially ordered example schedule.

Table 4 shows the results for each tick. After a certain start-up delay all operators receive and produce results in parallel. The parallel execution is considerably faster than the sequential execution which used as many ticks as it needed operations.

3.2 Universal Parallelism Strategy (UPS)

UPS adapts a strategy proposed for relational DBMS to overcome the above load balancing problems for query optimization [MOW97]. In the conventional
parallelization paradigm operators are interpreted as processes, and stream entries as passive data. In UPS we interpret each stream entry as a process, and the operator tree as passive data. UPS requires a shared everything architecture. The operator tree information is stored in shared memory, and is globally known to each process. For the model, each stream entry is a process. It reads the universal tree information and performs all operations along its path from a leaf to the root node.

Of course, a realistic implementation cannot evaluate each event as a single process, but requires a suitable organization of processes and resources as proposed in [MOW97]. For this paper, we do not discuss resource organization but the general idea of PPS. For the three operator categories execution works as follows:

- If a process performs a selector operator, the process checks the event information. If the selection predicate is fulfilled the process resumes for this entry. If not, it terminates.
- Constructor operators are performed symmetrically from both sides — called left and right — of the input streams. If a process performs a constructor operator coming from input stream right it creates a data entry and appends it to the buffer of right. Next, it searches the buffer for left for partners, creates new processes for each matching pair, and terminates. As an example, AND combines all entries in one buffer to each new entry of the opposite operand, and vice versa.
- If a process performs a collector operator, it passes the collector and resumes.

UPS achieves both pipelining and intra-operator parallelism, because each entry process performs all operators. Between operators the streams are used in a pipelining manner. The fine granularity overcomes the problems of the PPS, because no operator is a bottleneck.

Example: we use the same scenario as in table 4. Each event is a process. Table 5 shows the content and position of an event process. The process for A.1 passes the LEAF(A) operator, and next performs the OR operator, which takes one tick. It is now in the output stream of the OR operator, therefore shown as

<table>
<thead>
<tr>
<th>index</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>AND</th>
<th>BEFORE</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A.1</td>
<td>B.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>C.5</td>
<td>D.5</td>
<td></td>
<td></td>
<td></td>
<td>A.1</td>
</tr>
<tr>
<td>3</td>
<td>A.9</td>
<td>C.8</td>
<td>D.7</td>
<td>C.5</td>
<td>D.5</td>
<td></td>
<td>A.9</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>(C.5, D.7, 7)</td>
<td>B.4, (C.5, D.5, 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>(C.8, D.5, 8)</td>
<td>B.4, (C.5, D.5, 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>(C.8, D.7, 8)</td>
<td>B.4, (C.5, D.7, 7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>(B.4, (C.8, D.7, 8))</td>
<td>B.4, (C.8, D.5, 8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>(B.4, (C.8, D.7, 8))</td>
<td>(B.4, (C.8, D.7, 8))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4. PPS example for ALL(OR(A, BEFORE(B, AND(C, D))))
"result". The result of AND at tick 4 is a set of events, all computed in parallel. That set proceeds further in parallel. That is why UPS needs less time than PPS which proceeds one tuple at a time. Especially for the ALL semantics, UPS will lead to substantial performance improvements and is a tempting technique.

3.3 Parallel Event Detection and Timely Order

Parallelization overcomes the synchronization delays of sequential execution. Unfortunately, parallel computation may corrupt the timely order of events. We cannot simply adapt parallelization techniques developed for parallel query evaluation in relational DBMS. There, the semantics is set oriented and the base relations are complete. In contrast, composite event detection is based on open — often infinite or silent — streams of events. Cardinality and frequency of incoming events is unpredictable. Timely order is essential:

- Some operators are based on timely order, e.g. BEFORE.
- The consumption mode CHRONICLE (and variations as described in [Kri94]) requires timely ordered input streams.
<table>
<thead>
<tr>
<th>Ticks</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>AND</th>
<th>BEFORE</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A.1</td>
<td>B.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>C.5</td>
<td>D.5</td>
<td></td>
<td></td>
<td>A.1</td>
</tr>
<tr>
<td>3</td>
<td>A.9</td>
<td></td>
<td>C.8</td>
<td>D.7</td>
<td>(C.5, D.5, 5)</td>
<td>(B.4, (C.5, D.5, 5), 5)</td>
<td>A.9</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>(C.5, D.7, 7)</td>
<td>(B.4, (C.5, D.7, 7), 7), (B.4, (C.8, D.7, 7), 7)</td>
<td>(B.4, (C.8, D.7, 8), 8), (B.4, (C.8, D.7, 7), 7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(B.4, (C.8, D.5, 8), 8), (B.4, (C.8, D.5, 5), 5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(B.4, (C.8, D.7, 8), 8), (B.4, (C.8, D.7, 7), 7)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.** UPS example for $ALL(OR(A, BEFORE(B, AND(C, D))))$

– Timely order is an important optimization precondition, e.g. FIRST produces the result immediately if the input stream is timely ordered.

Therefore we have to find a strategy that respects the timely order of events, if necessary. We now discuss the behavior of PPS and UPS regarding timely order in more detail. As a proposition, we assume the input to the leaf operators to be timely ordered. Generally, we observe the following in operator trees:

1. The structure of the operator tree is unbalanced, i.e. the path lengths differ significantly. We do not assume that processing takes no time, therefore each operator along a path adds to the computation delay. Entries of short subtrees may overtake entries of deep subtrees.
2. The entry load is unbalanced. Due to the open streams, frequency of entries differ among streams. Some constructor operators have to compute more combinations than others. The delay allows other streams to overtake.

**PPS and Timely Order.** PPS fills and processes streams one tuple at a time. As a consequence, the detection process along one path in the tree is order preserving. However, streams among each other are not synchronized. For reasons described in (1) and (2), events may overtake. For example, A.9 arrives at the OR operator ahead of others with older time stamps (see Table 4, tick 4). As a consequence, the output streams of binary operators will not be ordered by time, unless we block all binary operators and sort the result before resuming.

**UPS and Timely Order.** In UPS both streams and operators are processed in parallel. Any given order will be destroyed by the execution. In addition to the problem of overtaking as in PPS (see Table 5, tick 4), operators are passed in parallel by several processes. For example, event processes for C.5, D.5, D.7, and C.8 perform the AND operation and subsequent operators along the path to the
root in parallel, that is, in unpredictable order, depending on resource load (see Table 5, ticks 4 to 6).

4 The SMILE Solution

We propose a combination of UPS if possible, and PPS if necessary. In order to cope with the problems of open streams, we introduce an extension to PPS, called heartbeats.

Not all the operators rely on timely order, and the ALL semantics for composite event detection is based on sets and produces sets of combinations without any order restrictions. Therefore UPS is a suitable parallelization strategy for ALL. The resulting compositions will be produced in arbitrary order, but quickly.

In composite event detection with CHRONICLE semantics the timely order of events is essential. The operands have to be sorted before producing results. Sorting might consume any performance improvement of the parallel execution. Since PPS is at least order preserving along paths in the operator tree, we may simply combine heads of timely ordered input streams. Unfortunately collector operators will not produce an ordered output because input streams may overtake each other. The OR operator has to sort the input before piping it to the output stream.

The sort itself is sensitive to the unpredictable behavior of open streams, which we call frequency skew: one operand is producing frequent results while the other does not. Collector operators have to wait for entries from the silent operand in order to decide which events can be sent to the output stream.

4.1 Hybrid Parallelism

We execute a detector tree according to its semantics either in UPS for ALL, or PPS for CHRONICLE. A detector might combine both semantics. In that case we differentiate the subtrees accordingly. Given a composition \( Y \) based on composition \( X \), we combine the strategies as shown in Table 6. The switch from one strategy to another is done by a special stream operator ACTIVATE which create a new process for each incoming entry (switch from PPS to UPS) and, symmetrically, a SORT operation that sorts incoming process data and transforms it to passive data (switch from UPS to PPS).

```
<table>
<thead>
<tr>
<th>ALL(Y)</th>
<th>CHRONICLE(Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL(X)</td>
<td>Y and X : UPS</td>
</tr>
<tr>
<td>CHRONICLE(X)</td>
<td>Y : UPS X : PPS</td>
</tr>
</tbody>
</table>
```

Table 6. Parallel strategies for ALL and CHRONICLE combinations
As stated before, unary operators are processed by PPS without any further extensions to the strategy. In order to produce sorted results for binary operators, we have to collect and sort entries in buffers before producing new results after comparing the time stamps. We propose sort-merge joins instead of naïve FIFO construction for constructor operators like BEFORE and AND.

In order to overcome the frequency skew problems discussed above, we introduce heartbeats as an extension of the PPS strategy. Heartbeats are dummy events comparable to dummy signals in network environments.

4.2 Heartbeats

A heartbeat is a special event of type H that is produced globally by the ADBMS itself. A heartbeat bears a time stamp as any regular event. The heartbeat type H is a subtype of all other types. For each CHRONICLE and OR combination in an event expression, copies of a heartbeat event are sent to the leaves of the subtree of OR operator. All operators accept heartbeats and send them to their output streams without further selection or construction.

Since streams in PPS in are timely ordered, the heartbeat H.t with time stamp t indicates that the entries in that stream preceding H.t are older or equal than t, while the entries following H.t are younger or equal than t.

As soon as a collector operator receives identical heartbeats from its input streams, the buffers are merged — including one copy of the heartbeat — and sent to the output stream. The buffers are flushed.

Example: based on the history of events: <A.1, A.2, A.5, A.8, A.11, A.13, A.14, A.15, B.16, ...> we want to detect compositions CHRONICLE(OR(A,B)) in a history of atomic events. Without heartbeat, all instances of A would have to wait for the first instance of B. In order to speed up the OR operator execution, we insert two global heartbeats to the history: <A.1, A.2, A.5, H.6, A.8, A.11, A.13, A.14, A.15, B.16, H.16, ...>.

Table 7 shows a section of the execution. Both leaves eventually receive and propagate H.6, see ticks 4 and 6. LEAF(B) sends the heartbeat as its first entry, since B.16 has not yet occurred. As soon as OR has received H.6, it merges all buffered entries into the output stream. Since the right operand buffer contains no entries of type B, it sends the buffered section of A.1, A.2, and A.5, followed by the heartbeat, see ticks 7 to 10. During the following period OR receives
and buffers instances from both A and B before the next heartbeat H.16. The buffers are merged without waiting for the heartbeats, and B.16 is sent along with instances of A.

The example illustrates that collector operators like OR now produce events at discrete heartbeat pulses. The heartbeat limits buffer space and enables the detector to produce possible event compositions as early as possible. In contrast to sequential detection, which immediately produces the first composition instance, the heartbeat pulse might delay composition instances for at most one heartbeat. But PPS then will produce results much faster than a sequential execution.

5 Related Research

At first view the issue of complex event detection with operator trees is closely related to the processing of time-series in temporal data base systems (cf. [SS87], [TCG+93]), or the processing of general sequence data (cf. [SLR96]). In those areas the timely order of data is essential. But temporal queries process completely stored data. This correspond only to composite event detection in a persistent history of past events, whereas the immediate detection of new composition instances from open streams is a new issue here.

Parallel processing in relational DBMS has been a research topic for a long time ([DG92], [Val93]). Relational DBMS are good candidates for performance improvements by parallelization because of the set-oriented nature of relations. The sequence-oriented second nature of time-series spoils set-oriented parallelization in temporal database systems. So, research for temporal databases is taking very first steps towards parallel processing ([KBO90], [MOW97]). The considered parallelization strategies are only feasible for stored data, and not for continuous processing as needed for composite event detection.

Most ADBMS approaches propose centralized, sequential detection of composite events. As an exception, Schwiderski et al. [SHM95] discuss time stamping and timely order of results in distributed composite event detection based on operator trees with the CHRONICLE semantics. The trees themselves are distributed across sites. The approach introduces two detection algorithms: a synchronized algorithm where operator nodes request for each input entry from other sites and an asynchronous algorithm where nodes accept input entries irrespective of timely order. The synchronized algorithm enforces timely order of results, which may lead to unpredictable delays due to failure of sites. In that case, delays may completely block the operator. Entries are buffered and merged comparably to our PPS approach. Parallelization itself is not the scope of this paper, but the distributed detection process leads to concurrent evaluation. In contrast, our approach assumes centralized event detection. We propose a parallel strategy suitable for the CHRONICLE and the ALL semantics, respecting timely order of events, and use both PPS and UPS.
6 Conclusion and Future Work

Our hybrid approach for parallel event detection is a useful optimization for applications where many trees have to cope with frequent and unsynchronized events. The paper discusses the parallelization with the help of a subset of event operator languages. The basic discussion of collectors, constructors, and selectors can be extended and applied to other ADBMS languages as well.

While UPS is suitable for the general ALL semantics, PPS with heartbeat extension is used for CHRONICLE semantics, as an example for time preserving parallelism. Our prototype implementation currently is based on the parallel programming language C-Linda [CG89], having a network of 6 SPARCstations with 8 processors in total. Both PPS and UPS are implemented for the discussed example operators.

Future work will investigate the dynamics of heartbeats in more detail. The frequency of heartbeats depends on elapsed time and the number of events already arrived at the buffers. The detector may dynamically increase the frequency if its buffers overflow, or decrease if the operators receive too many heartbeats. We may produce more than one global heartbeat for all detectors and use individual heartbeats for each disjoint OR subtree in the forest. This allows us to adjust the heartbeat frequency to the behavior of individual subtree streams.

References


This article was processed using the \LaTeX{} macro package with LLNCs style