Events and Streams: Harnessing and Unleashing Their Synergy!

Sharma Chakravarthy
Information Technology Laboratory (IT Lab)
Computer Science and Engineering Department
The University of Texas at Arlington, Arlington, TX 76019
Email: sharma@cse.uta.edu
URL: http://itlab.uta.edu/sharma
Acknowledgments

- This presentation is based on the work of many of my students, especially Dustin Jiang, Raman Adaikkalavan, Altaf Gilani, Satyajeet Sonune, Vamshi Pajjuri, Bala Kendai and Vihang Garg

- National Science Foundation for their support of MavEStream and other projects
This Tutorial tries to answer

- What is event-based active capability?
  - Now called complex event processing or CEP
- Early days – Driven by Databases/Applications!
- Resurgence – Driven by Stream processing!
- Stream Processing – need based on sensor and other data generation mechanisms
- How do we manage and exploit this synergy?
- How do we keep the complexity manageable and prevent these systems becoming programming projects?
Outline

- New Application and their requirements
- Need for both stream and event processing
- The premise of our approach
- Why it is Synergistic?
- Event processing
  - Early work -- Details
  - More recent work
- Stream processing
  - How is it different from DBMSs
  - Role of Data stream management systems (DSMSs)
  - Differences between the two
  - Related work
Outline (2)

- DSMS Issues (summary)
  - Architecture
  - Query processing
  - Modeling
  - Scheduling Strategies
  - Load Shedding

- Integration of Stream and Event Processing
  - Issues
  - Architecture
  - Example

- Conclusions
- References
Book Coming Out

Fundamentals of Data Stream Managements Systems
By
Sharma Chakravarthy and Dustin Jiang
To be Published by Springer, Fall 2008
MavEStream: Transforming Pervasive Real-time Data Into Actionable Knowledge
Stream Processing

MavEStream: Transforming Pervasive Real-time Data Into Actionable Knowledge

Monitoring Agencies

Emergency Services

Command Center

Sharma Chakravarthy: DEBS 2008
Complex Event Processing

Rules

MavEStream: Transforming Pervasive Real-time Data Into Actionable Knowledge

Environment Data  Traffic Data  Battlefield/Surveillance  Network traffic/routing data

Monitoring Agencies  Emergency Services  Command Center

Sharma Chakravarthy: DEBS
2008
Stream Processing

Complex Event Processing

Notification, Alerts, Rules

MavEStream: Transforming Pervasive Real-time Data Into Actionable Knowledge
Need for Monitoring

- Detect an Accident
- Detect Intrusion
- Traffic Monitoring
- Detect Collision
- Detect Pattern on Stock Feed
- Financial Applications

Stream proc + Event Processing

- If temperature > 50 turn on AC
- Aircraft Sensors

• Computation of stream data
• Monitoring on the output of stream computation
Homeland Security

Missile Defense program
real-time continuous tracking data

Load data to dbms: 1 minute
Compute its track: 30 seconds

1 minute to reach its target

Source: Ballistic Missile Defense Organization, Boeing Co.

From: http://www.fas.org/spp/starwars/program/nmd/
New Applications: Battlefield Monitoring

Command & Control Center

soldierID, location, heartBeat, bloodPressure
Traffic Monitoring

- Accident Detection:
  - Car becomes immobile
  - **Followed By** Another car in same segment reduces speed >30%
  - Action
    - Notify nearest police about the accident
    - Notify all cars in upstream
    - Notify toll station so waiting cars are not extra tolled.

- Is using only computation without casting it in terms of events better?
- Is it easy to understand and easy from the point of view design and management?
Traffic Monitoring
(our solution)

Accident Detection:
- Car becomes immobile
- **Followed By** Another car in same segment reduces speed >30%
- Action
  - Notify nearest police about the accident
  - Notify all cars in upstream
  - Notify toll station so waiting cars are not extra tolled.

Event Processing
- Identify cars which are in the same location for 2 mins
- Identify cars which reduced speed by >30% in 2 mins

Stream Processing
- Followed By Condition
  - Car in same segment

Sharma Chakravarthy: DEBS
2008
Integrated model
Advantages of Integrated Architecture

- Clear separation of events and prior processing (that gives rise to well-defined events)
- Prior processing can be stream or database or any other kind of processing (e.g., web services in distributed environments)
- Separation and attachment of multiple rules to events
- Re-use and minimization of complex events
- Clear use of complex events
Talk Outline

- What is event-based active capability?
Traditional View

Traditional System

Query

Updates Transactions applications

Repository

User-Driven

PULL Paradigm

Answers

July 2, 2008
Active Technology View

Self-monitoring / Reactive System

Query

Answers

Business rules
constraints
Invariants
situations to
monitor

System-driven

PUSH paradigm

Updates
Transactions
applications

Repository

July 2, 2008
Push Paradigm

Business rules constraints
Invariants
situations to monitor

Changes, Inserts/deletes

Self-monitoring / Reactive Application

Query

Optional Repository

PUSH paradigm

Answers
ECA Paradigm as the Basis for Active Functionality

- Ability to monitor the state of a database/application is fundamental for supporting active capability.
- The above observation lead to the development of the ECA paradigm in the HiPAC project.
- Monitoring the state of the database/application is implemented using ECA rules framework.
- An ECA Rule consists of:
  - an event
  - one or more conditions
  - an action
  - other rule/event attributes, such as coupling mode, priority, event consumption mode, grouping
Pioneers

- HiPAC – concentrated on events as well as integrating rules with all aspects of a DBMS (knowledge model, execution model, optimization, performance, recovery)

- POSTGRES – introduced rules in the context of Postgres DBMS

- ETM – enforced constraints in design databases using triggers and events
  
  (Remembering Klaus Dittrich)
Evolution of ECA Paradigm

Work in several disparate areas has come together to make the ECA rule paradigm:

- PL Exceptions
- Conditions/constraints
- Production rule/deductive systems
- Workflow Systems
- Coperative problem solving
- Service for distributed computing
History of Active DBMSs

UP to 1980
- ON Conditions (CODASYL) and Exit procedures (IMS)
- Invariants and triggers in System R
- Monitoring database state (Buneman and Clemens)

1981 to 1989
- Rules as a unifying paradigm: constraints, views, …
- Morgenstern introduces the term “Active Database” in 1983
- HiPAC formulates the ECA rule abstraction
- Event-trigger mechanism for CAD Databases (Dittrich and Kotz)
- Postgres Rule system
- Interbase and Sybase introduce triggers

1990 to 2001
- Research emphasis shifts to Active OODBMSs
- Prototypes: Sentinel, SAMOS, REACH, NAOS, ACOOD, Chimmera,…
- Almost all commercial relational systems support triggers
History of Active DBMSs

- **Beyond 2001**
  - Research emphasis shifts to Stream Processing
  - Stream processing systems include some event processing to support monitoring
  - A number of New systems commercial systems have been developed under the banner of stream, CEP, and mostly stream/CEP systems

- **Stream/CEP systems:**
  - StreamBase (based on Aurora)
  - Aleri
  - Amit
  - Esper
  - Coral8 (based on STREAM)
  - RuleCore (uses Snoop operators)
  - MAvESstream (based on Snoop and MavStream)
  - Weblogic
  - TIBCO
Active DBMSs

1. (Extended) relational active databases
   - Ariel
   - ETM
   - Postgres
   - Star bust
   - Informix, Ingres, Interbase, Oracle, and Sybase...

2. Object-oriented active databases
   - ACOOD
   - ADAM
   - Chimera
   - EXACT
   - HiPAC
   - NAOS
   - Ode
   - REACH
   - SAMOS
   - Sentinel
   - Triggs
Talk Outline

- Early days – Driven by Databases/Applications
Benefits of the ECA Paradigm

- Alternatives:
  - Ops X systems use CA format
  - Triggers in commercial databases use EA format
  - Active databases proposed ECA format

- Different Roles:
  - Event specifies **when** to check
  - Condition specifies **what** to check
  - Action specifies **how** to take care of the triggered situation
Event specification

An Event Specification Language needs to define
- Events, Event expressions, and Event modifiers
- Primitive events, and Event Operators
- Semantics of primitive events and event operators
- Semantics of event consumption modes
- Detection algorithms and their storage/computational complexity

- Note: in this period, it was assumed that a system would generate an event. There was no concept of stream processing at that time.
Events and Conditions

A Primitive Event
- has no duration
- may precede or follow another event
- may be unrelated to another event

On the other hand,

A Condition
- is a Boolean function
- is usually valid over an interval of time
- can be used to define a database state
Event Definition

An **EVENT** is an atomic (happens completely or not at all) occurrence e.g. Withdrawal of cash from bank

**Logical Event:** Specification of an event at the conceptual level

**Notion of Time:** Equi-distant and Discrete

**Physical Event:** Represents actual point of event occurrence in the system
Snoop Events

- **Event** – Any occurrence of interest

- **Simple Event**
  - Occurs at a point in time (e.g., card swipe)
  - Detected atomically at a point in time

- **Complex Event**
  - Combination of one or more events using an event operator
  - Occurs over an interval, detected at the end time

- **Event Expression**
  - Combination of simple and complex events using event operators

- **Event Parameters** – *domain-based*
Snoop Event Hierarchy

Event

- Primitive (Simple)
  - Temporal
  - External (abstract)
- Composite (Complex)
  - AND
  - NOT
  - SEQ
  - Plus

Formal Semantics

- Domain-based
- Relations DBMS
- Class-level
- Insert
- Update
- Delete

Object-Oriented

- Instance-level

Class-level

Object-Oriented Relations DBMS

Class-level

July 2, 2008
Sharma Chakravarthy: DEBS 2008
Primitive Events

- Occur @ a POINT

- e.g., \( f(\text{end_of_insert}) \rightarrow \) immediately after insert operation

- Event occurring at \((1,1)\)

  \[
  \text{Start-of-interval} = \text{End-of-interval}
  \]
Composite Events

- Constructed from the Primitive Events by means of set of *Snoop Operators*
- Composite Events occur over an INTERVAL
- Detected at a point in point-semantics

Event E1 occurs at a point (1,1), Event E2 occurs at a point (4,4), so that the Composite Event (E1^E2) starts at the point 1 and ends at the point 4.

We say E1^E2 occurs over an interval (1,4)
**Event Notation**

\[ e_{i}^{j} \]

- \( e \) – event
- \( i \) – event type
- \( j \) – instance of the event of type \( i \)

Examples: \( e_{1}^{1} \ e_{2}^{1} \ e_{1}^{2} \ e_{2}^{2} \)

**Event Tree:**

- **Leaf Nodes:** Primitive Events
- **Internal Nodes:** Composite Events
  - consists of primitive/ composite events with operators
Sequence

- Sequence of two events $E_1$ and $E_2$ is denoted by $E_1; E_2$ or by $E_1>>E_2$
- This means that the end time of occurrence of $E_1$ is guaranteed to be less than the start time of occurrence of $E_2$
- Example

<table>
<thead>
<tr>
<th>Events</th>
<th>Start</th>
<th>End</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$E_2$</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
| $E_1>>E_2$ | 1   | 2    | If $(\text{End (E}_1) < \text{Start (E}_2))$ then $\text{Start (E}_1; E_2) = \text{Start (E}_1)$  
$\text{End (E}_1; E_2) = \text{End (E}_2)$ |
Conjunction (AND)

- Conjunction of two events E1 and E2 is denoted by E1^E2.
- This composite event is detected disregarding the order of occurrence.
- Example: This would not be a sequence.

<table>
<thead>
<tr>
<th>Events</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>E1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>E1^E2</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

E2 starts the composite event detection and E1 ends the event.
Snoop Operators

- **OR (\(\lor\))**: Disjunction of two events \(E_1\) and \(E_2\), denoted by \(E_1 \lor E_2\), occurs when \(E_1\) occurs or \(E_2\) occurs.

- **AND (\(\land\))**: Conjunction of two events \(E_1\) and \(E_2\), denoted by \(E_1 \land E_2\), occurs when both \(E_1\) and \(E_2\) occur, irrespective of their order of occurrence.

- **SEQ (\(\gg\))**: Sequence of two events \(E_1\) and \(E_2\), denoted by \(E_1 \gg E_2\), occurs when \(E_2\) occurs provided \(E_1\) has occurred.

- **NOT (\(\neg\))**: The Not operator, denoted \(\neg(E_2)[E_1,E_3]\), detects the non-occurrence of the event \(E_2\) in the closed interval formed by \(E_1\) and \(E_3\).

- **PLUS (+)**: Sequence of an event \(E_1\) after a time interval \(T_I\), denoted \(E_1 + [T_I]\) occurs when \(T_I\) time units are elapsed after \(E_1\) occurs.
Snoop Operators

- **Aperiodic (A):** An Aperiodic event is denoted as A(E1, E2, E3). The event A is signaled each time E2 occurs during the half-open interval formed by E1 and E3.

- **Aperiodic Cumulative (A*):** A cumulative variant of A expresses as A*(E1, E2, E3). The event A* is signaled when E3 occurs and accumulates the occurrences of E2 in the half-open interval formed by E1 and E3.
Snoop Operators

- **Periodic (P):** A Periodic event is defined as an event that repeats itself within a constant and finite amount of time and is denoted as $P(E_1, E_2, E_3)$. $P$ occurs for every amount of time specified with the time string of $E_2$ in the half-open interval formed by $E_1$ and $E_3$.

- **Periodic Cumulative (P*):** $P^*$ is a cumulative variant of $P$ and is denoted as $P^*(E_1, E_2, E_3)$. $P^*$ occurs only once when $E_3$ occurs and accumulates the time $o$ occurrences of the periodic event whenever $E_2$ occurs.
Detection-Based Semantics

Events are detected at a point

<table>
<thead>
<tr>
<th>Events</th>
<th>Occurs</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>Point</td>
<td>Point</td>
</tr>
<tr>
<td>Composite</td>
<td>Interval</td>
<td>Point (last constituent event)</td>
</tr>
</tbody>
</table>
Snoop Operator Semantics (IB)

- Occurrence of an event $E$ (either primitive or composite) is a function from the time onto the Boolean values, True and False

$$E : T \times T \rightarrow \{\text{True, False}\}$$

Given by

$$E(t_{\text{start}}, t_{\text{end}}) = \begin{cases} 
\text{T(ue)} & \text{if an event of type } E \text{ occurs with } (t_{\text{start}}, t_{\text{end}}) \\
\text{F(alse)} & \text{otherwise}
\end{cases}$$

Primitive Events: $t_{\text{start}} = t_{\text{end}}$
Composite Events: $t_{\text{start}} < t_{\text{end}}$
Interval-Based Semantics

Events detected over an Interval

<table>
<thead>
<tr>
<th>Events</th>
<th>Occurs</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>Point</td>
<td>Interval $[t, t]$</td>
</tr>
<tr>
<td>Composite</td>
<td>Interval</td>
<td>Interval $[t_s, t_e]$</td>
</tr>
</tbody>
</table>

$t_s \rightarrow$ Start interval of the first constituent event
$t_e \rightarrow$ End interval of the last constituent event
Need for an Interval-Based Semantics

Why over an interval?

Input: $e_2, e_1, e_3$

Point Semantics

$e_1 >>> (e_2 >>> e_3)$

Interval-Based Semantics

$e_1 >>> (e_2 >>> e_3)$

Since $2 < 3$ Sequence is detected

Since $2 < 1$ is false, this is not a sequence

Sharma Chakravarthy: DEBS 2008
Snoop Operator Semantics (Contd.)

- **OR(∪):** Disjunction of two events E1 and E2, denoted by \( E1 \cup E2 \), occurs when E1 occurs or E2 occurs.

  Formally
  \[
  O (E1 \cup E2, [t_1, t_2]) = [O (E1, [t_1, t_2])] \lor [O (E2, [t_1, t_2])]
  \]
Snoop Operator Semantics (Contd.)

- **SEQ(;)**: Sequence of two events E1 and E2, denoted by E1;E2, occurs when E2 occurs provided E1 has occurred. Formally,

\[
O (E1; E2, [t_1, t_2]) =
\exists t, t' (t_1 \leq t < t' \leq t_2 \land
O (E1, [t_1, t]) \land
O (E2, [t', t_2]))
\]
Snoop Operator Semantics (Contd.)

- **AND(Δ):** Conjunction of two events E1 and E2 denoted by E1 \( \land \) E2, occurs when both E1 and E2 occur, irrespective of their order of occurrence.

\[
O (E1 \triangle E2, [t_1, t_2]) = \\
\exists t, t' \ (t_1 \leq t \leq t_2 \land t_1 \leq t' \leq t_2 \land \\
((O (E1, [t_1, t]) \land O (E2, [t', t_2])) \lor \\
(O (E1, [t', t_2]) \land O (E2, [t_1, t])))
\]
Event Consumption Modes

- Event Consumption Modes or Parameter Contexts

- **Recent** – multiple occurrences of the same event only refine the previous value

- **Continuous** – events are detected along a moving time window

- **Cumulative** – multiple occurrences of constituent events are grouped

- **Chronicle** – correspondence between constituent events and their occurrences are maintained
Snoop Parameter Context/ Event Consumption Modes

- If occurrence of any of the primitive events makes a composite event occur, then there is no need for a Context. Example: disjunction, Aperiodic, absolute events.
- If a composite event expression requires an interval for its detection, then the notion of Context comes into picture. Example: Conjunction, Sequence.
- Parameter Contexts (or event consumption modes) are functions that maps a global event history to a composite event history:
  - Unrestricted
  - Recent
  - Chronicle
  - Cumulative, and
  - Continuous
Event consumption modes

- Used for restricting the event occurrences in the unrestricted context to obtain a meaningful subset
- Choice of event consumption mode also affects the complexity of composite event detection and storage requirements
- Initiator (Start-of-Interval) -- a primitive event that starts the interval for the occurrence of a composite event
- Terminator (End-of-Interval) -- a primitive event that ends the interval for the detection of a composite event
- For a primitive event, the initiator also acts as its terminator
Event consumption modes

- Unrestricted/General
- Recent Context
  Events happening at a fast rate and multiple occurrences of the same event only refine the previous value
  e.g., sensor applications, tracking applications
- Chronicle context
  Different types of events have an established relationship between their occurrences
  e.g., begin and end of a task, bug-report and bug-fix
Event consumption modes

- **Continuous context**
  Composite event detection along a moving time window
  e.g., trend analysis, stock market

- **Cumulative context**
  Multiple occurrences of constituent events need to be grouped and used in a meaningful way
  e.g., banking applications
Composite Event Detection

- RECENT CONTEXT
  - Events happening at a fast rate and multiple occurrences of the same event only refine the previous value
    - e.g., sensor applications, tracking applications
  - Only most recent initiator is used
  - When a composite event occurs all the occurrences of constituent events that cannot be initiators in the future are deleted.
  - Not all occurrences of a constituent event will be used
  - An initiator will continue to initiate new event occurrences until a new initiator occurs.
Composite Event Detection

- **Chronicle context**
- Different types of events have an established relationship between their occurrences.
- e.g., begin and end of a task, bug-report and bug-fix.
- For an event occurrence initiator & terminator pair is unique.
- Oldest initiator is paired with the oldest terminator (chronological order).
- Constituent events cannot occur in any other detection of the same composite event.
Composite Event Detection

- **Continuous context**
- Composite event detection along a moving time window
- e.g., trend analysis, stock market
- Each initiator starts the detection of that composite event.
- A single terminator may detect one or more occurrences of that same composite event.
- Both initiator and terminator are discarded after the event is detected.
- An initiator will be used at least once to detect that event.
**Composite Event Detection**

- **Cumulative context**
  - Multiple occurrences of constituent events need to be grouped and used in a meaningful way.
    - e.g., banking applications.
  - All occurrences of an event type are accumulated as instances of that event until the event is detected.
  - Once detected all constituent event occurrences are deleted.
  - An event occurrence does not participate in two distinct occurrences of the same composite event.
  - This context cannot be generated as a subset of the event-history generated by the unrestricted context.
Composite Event Detection

- **Disjoint Specification**
  - Involves detection of composite events over a non-overlapping time interval.
  - For any 2 occurrences of a particular composite event
    - \( T_{\text{end}}(\text{initiator\_event}) < T_{\text{start}}(\text{terminator\_event}) \)

\[ \begin{array}{c}
1,8 \\
\wedge \\
1,4 \\
E1 \\
1,8 \\
\wedge \\
1,4 \\
E1 \\
5,8 \\
E2 \\
\end{array} \]  
\[ \begin{array}{c}
1,8 \\
\wedge \\
1,4 \\
E1 \\
3,8 \\
\wedge \\
1,4 \\
E1 \\
\end{array} \]  

- \( T_{\text{end}}(\text{initiator\_event}) = 4 \)
- \( T_{\text{start}}(\text{terminator\_event}) = 5 \)
- \( (4 < 5) \) **Disjoint Event**
- \( T_{\text{end}}(\text{initiator\_event}) = 4 \)
- \( T_{\text{start}}(\text{terminator\_event}) = 3 \)
- **Over Lapping Events**
Generalization of Composite Events

- Snoop, SnoopIB and other event specification languages support only \textit{time-based semantics} for complex event operators.

- User activities over a period of time can be characterized as event patterns and can be captured using complex events.
  - For a composite event, tracking of the constituent events raised by the same user requires $Eexpr$.

- Similar to simple events, composite event operator semantics must be defined with $leexpr$ and $Eexpr$. 
Motivations – Simple Events

Event $E = f (\text{instance}, \text{parameters})$;

- `setPrice(price);`
  - Event $ES1 = setPrice(price);$  
  - Event $ES2 = setPrice(stockId = “GOOG”, price);$  
- When Occurrence of Interest is  
  - Price $\theta$ $100$ *(explicit parameter)*  
  - stockId = “GOOG” AND Price $\theta$ $500$ *(implicit and explicit parameter)*  
  - Events that occur only after 18.00 hrs everyday *(implicit parameter)*

- $\theta \rightarrow <, >, \leq, \geq, \neq, =, \in, \ldots$
- Snoop cannot handle
Generalizing Primitive Event Definition

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iexpr</td>
<td>Expression on Implicit parameters that returns Boolean</td>
</tr>
<tr>
<td>Eexpr</td>
<td>Expression on Explicit parameters that returns Boolean</td>
</tr>
<tr>
<td>Expression based on</td>
<td>&lt;, &gt;, &lt;=, &gt;=, !=, =, ∈, ...</td>
</tr>
</tbody>
</table>

Event $E = (\text{setPrice}(\text{price}), (\text{Iexpr} \land \text{Eexpr}))$;

Event $ES1 = \text{setPrice}(\text{price})$;
Event $ES2 = (\text{setPrice}(\text{price}), (\text{price} > 100))$; /*Eexpr*/
Event $ES3 = (\text{setPrice}(\text{price}), (\text{t_occ} > 18.00))$; /*Iexpr*/
Event $ES4 = (\text{setPrice}(\text{price}), (\text{stockId} = \text{“GOOG”}))$; /*Iexpr*/
Event $ES5 = (\text{setPrice}(\text{price}),$

$((\text{stockId} = \text{“GOOG”}) \land (\text{price} > 500))$); /* I,E */
Generalizing Event Operator Semantics

\[ E: ((\text{Eop} (E_1, \ldots, E_n), (\text{Iexpr} \land E_{\text{expr}})), [t_s, t_e]) \]

- **Eop** → AND, OR, Sequence, Plus, NOT, ...
- **E_1, \ldots, E_n** → Simple and Complex named events
  - \( n \) can be 1 (unary), 2 (binary), 3 (ternary), ... for event operators
- **t_s** → Start time of the event (generally \( E_1 \)'s start time)
- **t_e** → End time of the event (generally \( E_n \)'s end time)
- **Iexpr**
  - Time, instance, and other implicit parameters
  - **Snoop** and **SnoopIB** use predefined Iexpr
  - It has default point- and interval-based semantics
- **E_{\text{expr}}**
  - Parameters of event’s \( E_1, \ldots, E_n \)
Event Pattern Example

Allow users *active* in any role **TO** enter “Pregnancy Ward” **FROM** “Virus Ward” **IFF** the user has made a “Hygiene Stop”
Simple and computed events

- Simple events are those that are provided by the underlying system
  - databases, applications, click stream, RFIDs, sensors, any kind of monitoring system

- Computed events
  - Primitive events explicitly defined by the user which requires computation
    - Continuous queries generating events
    - Mining computation that generates events

- The above are needed for defining higher level events from raw data
  - e.g., traffic congestion, accident, pollution, fire, …
Approaches for Incorporating Active Technology

- Integrated
  - for NEW systems – Sentinel

- Wrapper-based
  - for LEGACY systems

- Distributed/Heterogeneous environments
  - GEM (Global Event Manager)

- Agent-Based/Mediated
  - with COTS (Sybase, Oracle, DB2, …)
Stream Processing
Database Approach

- PROGRAM 1
- PROGRAM 2
- DBMS
  - Query Processor
  - Transaction Mgr
  - ...
- Integrated Database
**Queries, Query Plans, and Operators**

**Query in English:**
Give me all movie names after 1979

**Query in SQL:**
```
SELECT Name
FROM Movies
WHERE Year > 1979
```
Advantages of a DBMS

- Data independence
- Efficient data access
- Data integrity & security
- Data administration
- Concurrent access, crash recovery
- Reduced application development time
Outline

- Database Management Systems (DBMSs)
- **New applications & their requirements**
- Role of Data stream management systems (DSMSs)
- Differences between the two
- Related work
- DSMS Issues
  - Architecture
  - Extensions to SQL
  - Query processing
  - Modeling
  - Scheduling Strategies
  - Load Shedding
- Integration of Stream and Event Processing
- MavStream System – design and implementation
- Conclusions
- References
So why not use a DBMS?

- Expensive/complicated to set up & maintain
- This cost and complexity must be offset by need
- General-purpose, not suited for special-purpose tasks (e.g. stream, text, XML processing!)
- Not good for applications that need real-time processing
Questions

1. Find all soldiers who need help
   - A soldier is in critical condition and needs help if his blood pressure is less than 60 and his heartbeat is less than 50.

2. Find all soldiers who need help and all possible helpers who are near by (within 100 meters)
   - Distance (wounded soldier, helper) is less than 100 meters.
Database Approach

1. Load all sensor (or stream) data into databases
2. Define triggers (or run applications) to detect conditions and trigger actions.
Problems with the Database Approach

1. Time to load sensor data into a database is large due to the large volume of sensor readings.
2. The old data in the database is not useful since we are only interested in current status.
3. The delay introduced by this approach can be large, which is not acceptable for time critical applications.
4. A large amount of triggers are not well supported by current database management systems.
New approach: Data Stream Management System

Input Data Streams

Query Plans

Feedback: Command & Control information
Summary

Current/earlier Approach
Stream Processing Approach
What’s a Data Stream?

A continuous data stream: a sequence of data items that are ordered by time or an attribute.

- Examples of data streams:
  - Readings from a sensor
  - Readings of stock price
  - Traffic packets over a link ($T_1$)
  - …

- It is considered **continuous** as monitoring is done over long periods of time
Where do data streams come from?

- **Computer Network Management**
  - Traffic data; for a OC192 link, traffic data flow can reach 250Mbytes/sec
  - SNMP data
  - Route table information, BGP table, routing forward table
  - Topology information and so on

- **Telecommunication System**
  - Call Detail Record information; Hundreds of millions of phone call records from ten millions of customers per day
  - Network management data: alarm message, performance data,…

- **Sensor System**: sensor readings
  - temperature, lights, others

- **Financial System**: transaction records.

- **Web system, health care and many others.**
Characteristics

- Stream Characteristics
  - Continuous stream
  - Unpredictable input characteristics
  - Unbounded in size

- Application Characteristics
  - Online processing
  - Real time response
  - Approximate results

- The above are different from traditional DBMS data and application characteristics
What is a Continuous Query (CQ)?

- The main computations over data streams are long-running continuous queries.
- A continuous query is issued once and logically runs continuously over data streams and traditional database relations.
Example 1 (CQs)

- Going back to the battlefield example,
  - Find the location of all solders who need helps immediately;

```sql
SELECT location, soldierID
FROM B  //a stream
WHERE B.heartBeat < 50
AND B.bloodPressure < 60;
```

- Output results as a data stream
Example 2 (Join)

- Find **soldiers** who need helps immediately and **all helpers** within 100 meters;

SELECT soldiers.soldierID, helpers.soldierID

FROM B as soldiers, B as helpers

WHERE Dist(soldiers.location, helpers.location) < 100 meters

  AND soldiers.heartbeat < 50

  AND soldiers.bloodPressure < 60

  AND soldiers.soldierID <> helpers.soldierID
More Continuous Queries

Incoming (call_ID, callee, time, event)

DSMS

Outgoing (call_ID, caller, time, event)

Central Office

Central Office

DSMS

event = START or END

John

May
Query 1 (join)

- Pair up **callers** and **callees**

  ```sql
  SELECT O.caller, I.callee
  FROM Outgoing O, Incoming I
  WHERE O.call_ID = I.call_ID
  ```

- Can provide **result as data stream**
- Requires **unbounded temporary storage**
Query 2 (group-by aggregation)

- **Total connection time** for each caller
  
  ```
  SELECT O1.caller, sum(O2.time - O1.time)
  FROM Outgoing O1, Outgoing O2
  WHERE (O1.call_ID = O2.call_ID
     AND O1.event = START
     AND O2.event = END)
  GROUP BY O1.caller
  ```

- **Cannot provide result in (append-only) stream**
  - Output updates?
  - Provide current value on demand?
Outline

- Database Management Systems (DBMSs)
- New applications & their requirements
- Role of Data stream management systems (DSMSs)
- Differences between the two
- Related work
- DSMS Issues
  - Architecture
  - Extensions to SQL
  - Query processing
  - Modeling
  - Scheduling Strategies
  - Load Shedding
- Integration of Stream and Event Processing
- MavStream System – design and implementation
- Conclusions
- References
## DBMS Vs. DSMS

<table>
<thead>
<tr>
<th>DBMS</th>
<th>DSMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistent relations</td>
<td>Transient streams</td>
</tr>
<tr>
<td>One-time (ad hoc) queries</td>
<td>Continuous queries</td>
</tr>
<tr>
<td>Random access</td>
<td>Sequential access</td>
</tr>
<tr>
<td>“Unbounded” disk store</td>
<td>Bounded main memory</td>
</tr>
<tr>
<td>Only current state matters</td>
<td>History/arrival order critical</td>
</tr>
<tr>
<td>Relatively low update rate</td>
<td>Possibly high/low data rates</td>
</tr>
<tr>
<td>No real-time services</td>
<td>Real-time requirements</td>
</tr>
<tr>
<td>Assume precise results</td>
<td>Approximate results</td>
</tr>
<tr>
<td>Transaction management</td>
<td>No transaction management</td>
</tr>
</tbody>
</table>
Main Memory DBMSs

- These were developed to overcome the impedance mismatch between I/O and cpu processing
- Main memory is used as storage instead of disk
- Instead of buffer (in traditional DBMSs), data is stored in memory
- Larger amounts of main memory are used

- Used in embedded applications to improve response time
Main Memory DBMSs

- There are several main memory DBMSs that have been developed
  - TimesTen
  - Polyhedra
  - DataBlitz

- All are traditional DBMSs, support SQL processing using main memory data.
- Typically, used for caching a subset of the relational database in main memory
Real-Time Transaction Processing

- Used the notion of traditional transactions and secondary storage
- The emphasis was on scheduling transactions to meet soft and hard deadlines
- Priority inversion and avoiding unpredictable delays introduced by secondary storage were main topic of interest
Embedded DBMSs

- Small footprint is important
- Transition from embedded to full-scale system is important
- Use of new devices such as flash and SSD
- Componentization seems critical for mixing and matching relevant features
Outline

- Database Management Systems (DBMSs)
- New applications & their requirements
- Differences between the two
- Role of Data stream management systems (DSMSs)
- Related work
- DSMS Issues
  - Architecture
  - Extensions to SQL
  - Query processing
  - Modeling
  - Scheduling Strategies
  - Load Shedding
- Integration of Stream and Event Processing
- MavStream System – design and implementation
- Conclusions
- References
DSMS in Sensor Network System

- Small, low cost battery powered microprocessors with 1 – 4 sensors
  - Light, temperature, vibration, acceleration, AC power, humidity.
- 10Kbit – 1Mbit wireless networks, 100ft range.
- “Ad-hoc” networking – no predefined routes.
DSMS in Distributed Web System

- Coordinate many distributed clickstream analysis
  - Track heavily accessed web pages

```
+-----------------+-----------------+-----------------+
| Personalization | Performance      | Load            |
|                 | Monitor          | Balancing       |
```

- Continuous Query
- Web Server
  - American
  - Asia
  - Europe
- Click Streams
- Load Balancing
DSMS in Finance System

- Evaluate queries over real-time streaming financial data
  - credit card swapping stream
  - stock tickers and news feeds

```
Fraud Detection System | Stock Query System | Load Balancing
```

```
Continuous Query
```

```
Stream Multiplex De-multiplex
```

```
Credit Card Swapping stream
```

```
Transaction Streams
```

Sharma Chakravarthy: DEBS 2008
Higher Level Questions

✓ What’s the difference between these two models?
✓ Can we have a general model to support both DBMS and DSMS?
✓ Do we need to build a general-purpose model, algorithms and system for data streams?
✓ Does a DSMS functionality implemented as part of a DBMS more efficient?
✓ Is DSMS a generalization of a DBMS?
Challenges of DSMS

- Unbounded memory requirements
  - Blocking operators
  - Synopses and statistic summarizations
  - Sliding windows

- Online processing
  - Need for queues

- Operator/query Modeling
  - Network of queues and servers
  - Queuing theory-based

- Query processing/Optimization
  - Push-based strategies
  - No cost-based model (surprisingly)
  - QoS driven (memory, latency, throughput)
DSMS Challenges (Contd.)

- **Scheduling Strategies**
  - Chain scheduling (for memory)
  - Path capacity (for latency, thruput)
  - Hybrid

- **Load Shedding**
  - Based on run-time monitoring
  - Change scheduling and/or activate load shedding
  - Bound on approximate answers

- **Admission control**
  - If everything else fails
DSMS Challenges (Contd.)

- System Design and Implementation
  - Operator execution
  - Buffer/queue management
  - Scheduling
  - How to monitor output as well as selectivity
  - Run-time optimizer
  - When and how much load to shed
  - Where to shed load?

- Integrating stream and event processing
  - Synergy between the two
  - Integration issues
### Related Work - Stream Processing

<table>
<thead>
<tr>
<th></th>
<th>Inputs</th>
<th>Optimization</th>
<th>Query Types</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Aurora</strong></td>
<td>GUI (blocks and arrows)</td>
<td>Superboxes</td>
<td>Ad-hoc CQ</td>
<td>Data Flow</td>
</tr>
<tr>
<td>[VLDB Journal, 2003]</td>
<td></td>
<td>Scheduling</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load shedding</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Streams</strong></td>
<td>Modified SQL</td>
<td>Scheduling</td>
<td>CQ</td>
<td>Data Flow</td>
</tr>
<tr>
<td>[Innovative Data System Research 2003]</td>
<td></td>
<td>Load shedding</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fjord</strong></td>
<td>N/A</td>
<td>Combined Queries</td>
<td>Ad-hoc CQ</td>
<td>Central Tuple Routing</td>
</tr>
<tr>
<td>[Data Engineering 2002]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MavStream</strong></td>
<td>GUI/ ASCII</td>
<td>Modeling</td>
<td>CQ</td>
<td>Data Flow</td>
</tr>
<tr>
<td>[SAC 2004, UTA Theses]</td>
<td></td>
<td>Scheduling</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load shedding</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Aurora Project

- Targeted towards stream monitoring applications
- Consists of large network of triggers (data-flow graph)
- Application administrators create and add triggers
- Compile-time and run-time optimization of trigger network
- Detects resource overload and performs load shedding based on application-specific measures of QoS
STREAM

- Targeted exclusively for stream monitoring apps
- Compile-time and run-time optimization
- Detects resource overload and performs load shedding
- Chain scheduling for optimizing memory
- Approximate answers
MavStream

- Holistic approach to stream processing
- Operator and query modeling
- Scheduling algorithms for QoS optimization
- Load shedding strategies
- Synergistic Integration and events with stream processing
Blocking operators

In a traditional database system, there are two kinds of operators:

- **blocking operators (need to see all data)**
  - sorting
  - Union
  - aggregation operators: sum, avg, count, min, max and others

- **Non-blocking operators (no need to see all data)**
  - Select, project, and others
Blocking operators Problems

- Problems
  - If the blocking operators are at the top, output is not produced
  - If they are in the middle (e.g., sort for a sort-merge join), rest of the computation is blocked

- Solutions
  - Replace the blocking operators by some kinds of non-blocking operators which can do similar function approximately
    - V. Raman, et al. developed juggle which can replace sorting operation by locally reordering a data stream, some tuples may be out of order
Window as a Solution

- Based on the Unit used for marking a window boundary.
  - Physical
  - Logical

- Based on the Length of the window
  - Snapshot Window
    - Fixed Window.
    - Query: *Count the number of people entering the lab from 9:00 am – 5:00 pm on Dec 13 2007*
  - Landmark Window
    - Starting point is fixed and end point is moving
    - Query: *List the usage of the devices in Room A from 9:00 am – 5:00 pm on Dec 03 2008 at the end of each hour*
Query Window Classification (contd.)

- Sliding Window
  - Both starting and end points of a window are moving

- Disjoint Sliding Window
  - Query: Display the average temperature of Room A every hour from 10 pm to 10 am on Dec 15 2007

- Overlapping Sliding Window
  - Query: Give a count of the number of devices turned off over one hour interval between 10 am – 11 am on Dec 15 2007 every 10 minutes
Query Window Classification (contd.)

- Reverse Landmark
  - Here the window expands in the reverse direction.

- Reverse Sliding
  - Here the sliding window expands in the reverse direction.

- Reverse windows make use of historical data which may not be supported in some stream systems.
Examples of Windowed Queries

- **Snapshot query:**
  - How many times the coffee maker was turned on in room “B” between 5 p.m. and 6 p.m. On Nov 25, 2007?

- **Landmark query:**
  - Continuously list the details of individuals entering in MavHome from 5 p.m onwards till 9 p.m on Nov 25, 2007 at the end of each hour.
Example of Windowed Queries

- **Disjoint Sliding Window Query:**
  - Select common devices that were turned “On” in two different rooms every hour between 5 p.m and 9 P.M on June 5 2007

- **Overlap Sliding Window Query:**
  - Display average temperature over one hour interval between 5 P.M. and 7 p.m on June 5, 2007 every 10 minutes.
MavStream Extensions

```
SELECT select_list
FROM from_list
WHERE where_clause
BEGIN begin_window (time /tuples)
END end_window
HopSize (lower_bound, upper_bound)
End Query
```

(lower_bound, upper_bound):
  (constant, constant) : Snapshot
  (constant, variable) : Landmark
  (variable, variable) : Sliding Window

- ve values provide reverse windows
MavStream example

- Begin and End Window specifies first window.
- Hop size provides a way to increment the window.
- End query provides a conditional end to windowed query.
- Example: Display the average temperature of Room A every hour from 10 am to 10 pm on Dec 15 2003.
  
  \[
  \text{Begin Window} = 10 \text{ am on Dec 15 2003.} \\
  \text{End Window} = 11 \text{ am on Dec 15 2003.} \\
  \text{Hop Size} = (1\text{hr, 1hr}) \\
  \text{End query} = 10 \text{ pm on Dec 15 2003.}
  \]
Query Processing

- Traditional query processing
- Pull Vs. Push Paradigm
- Need for queues in between operators
A traditional query processing employs a standard pipelined (or iterator-based) approach.

A traditional query plan has a tree shape and is executed top-down in a “pull” style.
Query processing: problems

What’s wrong with a pull style query execution plan:

- It may lose data because data arrives in a bursty manner.
- It may be blocked because of the slow data arrival rate.
- Binary operators like Join, union and so on, may not work properly over two or more different data streams because of different data arrive rate.

```
Join

Data stream 1
10Mbytes/s
Or so

Data stream 2

Join

5Kbytes
Or so
```
Query processing: proposed solution

- A query execution plan consists of operators connected by queues.
- Operators can maintain intermediate state in a data structure locally (synopsis).
- All operators are operated over a window specified for each query.

A query plan based on queue
Continuous Query Plan

- Synopses
- Running Op
- Ready Op
- Waiting Op

Scheduler

Input Data Streams

Query Plans
Cost Model

- Surprisingly there is no QoS modeling of CQs!
  (except Naughton and Viglas work which is slightly different from QoS modeling)
- Traditional DBMS cost models do not work for continuous queries
- There is no I/O (traditional models are i/o based!)
- Traditional models do not consider QoS
  - Tuple latency
  - Memory usage
  - Throughput (bursty or smooth)
Why Model Continuous Queries

1. **System Capacity Planning**
   - Given a set of continuous queries over data streams, we would like to know what kind of system, in terms of resources, is able to process the given CQs?

2. **Facilitate QoS Delivery Mechanisms**
   - Provide quantitative information to choose different QoS delivery mechanisms, such as different scheduling strategies, load shedding, and so on.

3. **QoS Verification**
   - Confirm whether the final results of a query really satisfy defined QoS requirements.
Overview of the Solution

1. **Model individual operators**
   ✓ Each operator is modeled as a queueing system.
   ✓ Find the tuple latency and the queue size.

2. **Model continuous queries**
   ✓ Both an individual continuous query and a multiple-CQ processing system are modeled as a network of queueing systems with vacation time and setup time.
   ✓ Find the tuple latency and the queue size.

3. **Experimental validation of our analysis**
Assumptions

1. The input process of an input data stream is assumed to be a Poisson distribution (inter-arrival interval forms an exponential distribution).

2. Properties, such as selectivity, processing capacity, and so on, of an operators are assumed to be known.

3. Internal queue size is unbounded (no tuple is dropped).

4. Final results are consumed by applications immediately (final results are NOT buffered).
A query plan based on queue

- A query execution plan consists of operators connected by a queue.
- Operators can maintain intermediate state in synopsis data structure locally.
- Some operators are computed over a window size.
Modeling Query Processing

- A query plan is modeled as a network of queueing system.
  - Input buffer => input queue
  - Functionality => service facility
  - Relationship of two operators => connection of two queueing systems

- We have to know
  - Input characteristics
  - Service characteristics
Window-based Hash Join

Cost of processing one tuple

1. Insert the tuple into Hash_S;
2. hash the tuple into the hash table Hash_R
3. search the corresponding bucket with the join condition;
4. output the matching tuples to Q_O if any.

\[
D_i = 2 \cdot C_H + C_O + C_E \cdot \left( \frac{H_j(n)}{m_j} \right) \begin{cases} j = r & \text{if} \quad i = s \\ j = s & \text{if} \quad i = r \end{cases}
\]
Modeling HASH JOIN

- There are two types of inputs for a join operator, left input and right input.
- Input rate \((\lambda_1 + \lambda_2)\), and service time \(D_1, D_2\)
- Hash Join is modeled as \(M/(D_1, D_2)/1\)
- Only steady state is analyzed
  - Transition phase is short and difficult to analyze.

There are two types of inputs for a join operator, left input and right input. The input rate is \((\lambda_1 + \lambda_2)\), and the service times are \(D_1\) and \(D_2\). Hash Join is modeled as \(M/(D_1, D_2)/1\). Only the steady state is analyzed, with the transition phase being short and difficult to analyze.
Mean queue size

- Given a window-based hash join system over two data streams with input mean rate 1; 2, and service times D1;D2 for the tuples from left stream 1 and right stream 2 respectively, the mean queue size under the steady state is given by:
Average Queue Size

\[
E[q] = \lambda_1 D_1 + \lambda_2 D_2 + \frac{(\lambda_1 + \lambda_2)(\lambda_1 D_1^2 + \lambda_2 D_2^2)}{2[1 - (\lambda_1 D_1 + \lambda_2 D_2)]}
\]

If we increase both the processing rate and input rate at the same time, the average queue size doesn’t change.

\[
\lambda_1' = k\lambda_1 \quad \lambda_2' = k\lambda_2 \quad D_1' = D_1 / k \quad D_2' = D_2 / k
\]

- Probability distribution of the number of tuples
- Average tuple latency
- Cumulative probability of tuple latency
Extension – one relation on local disk

- If the local relation can be fit into main memory, then the service times for both local relation tuple and stream tuple are constants.
- If it cannot be fit into main memory, the service times are still constants except the service times times are longer.
- M/D/1 Queueing system.
Experiment Setup for Modeling Operator

- Input streams: Poisson Data Streams
- Operator: Hash-Join Operator
- Dual-Processor Alpha machine

Measure:
1. Queue size and waiting time in the queue
2. PDF of queue size
3. CDF of waiting time in the queue
Modeling Operator: CDF

Mean Tuple Latency(s)

- Theoretical
- Experimental

Mean Queue Size

- Theoretical
- Experimental

Cumulative Probability

System Load

Sharma Chakravarthy: DEBS
2008
Modeling Continuous Queries

CASE a: queuing system with external input(s)
CASE b: queuing system with internal input(s)
CASE c: queuing system with internal and external inputs
Modeling Continuous Queries (2)

- For each operator, once it gains the processor, if its input queue is empty, the operator goes to vacation immediately. Otherwise, the processor needs a setup time and then serves a certain number of tuples. Then the operator goes to vacation (processor serves other operators or process other tasks).

- Each operator is modeled as a queuing system with setup (U) and vacation time (V)
The scheduling strategy determines when an operator can get the processor.

The service discipline determines how many tuples it can serve once an operator gains processor.

- **Gated-service discipline**
  - Only serve the tuples waiting in its queue when an operator gains processor, (tuples arrive during setup time and serving time are postpone to next round)

- **Exhaustive-service discipline**
  Until queue is empty
CASE a: Modeling Operators with external inputs

- External input is a Poisson distribution.
- M/G/1 queuing system with setup time and vacation time.

Under both service disciplines, find:
- The Average waiting time of a tuple at this operator
- The Average queue size at this operator
CASE b: Modeling operators with internal inputs

- **Input process is the output process of another operator.** Its input process is not a Poisson process any more. However, the number of tuples arriving during its vacation time is equal to the number of tuples output by another operator.

- **The results under the gated service discipline are the same as those under the exhaustive service discipline.**

- **We need to find the average queue size, and the average tuple latency.**
Case c: Modeling Operators with internal and external inputs

- One input is a Poisson distribution, another is not, which can be considered as a combination of case a and case b.

- Its busy period, average queue size and average tuple latency are weighted sum of two inputs (one for case a, another for case b).
Setup for Modeling CQs

- Input streams: Poisson input data streams
- Queries: 16 CQs with 116 operators

- Measure: (for a CQ)
  - Waiting time at each operator including waiting time in the queue.
- Did a set of experiments
- Show the best case and the worst case

Compare the results from Poisson streams and Bursty (self-similar) streams.

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Processing Rate(#/s)</th>
<th>Left Selectivity</th>
<th>Right Selectivity</th>
<th>Setup Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O5</td>
<td>5894</td>
<td>0.308514</td>
<td>0.3007712</td>
<td>7.44224E-4</td>
</tr>
<tr>
<td>O4</td>
<td>28461</td>
<td>0.36534</td>
<td>-</td>
<td>1.1502E-5</td>
</tr>
<tr>
<td>O3</td>
<td>5285</td>
<td>0.30703</td>
<td>0.300177</td>
<td>6.9477E-4</td>
</tr>
<tr>
<td>O2</td>
<td>21861</td>
<td>0.487534</td>
<td>-</td>
<td>1.9599E-5</td>
</tr>
<tr>
<td>O1</td>
<td>41684</td>
<td>0.24288</td>
<td>-</td>
<td>4.8044E-5</td>
</tr>
</tbody>
</table>
### Table 2. Tuple Latency (Seconds) Under **Hierarchical** Scheduling Strategy

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>O1</th>
<th>O2</th>
<th>O3 (10^-3)</th>
<th>O4</th>
<th>O5 (10^-3)</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical</td>
<td>0.14690</td>
<td>0.147114</td>
<td>2.3277</td>
<td>0.146988</td>
<td>2.64285</td>
<td>0.15111</td>
</tr>
<tr>
<td>Best Exper.</td>
<td>0.14868</td>
<td>0.149151</td>
<td>2.4561</td>
<td>0.14665</td>
<td>2.5963</td>
<td>0.15127</td>
</tr>
<tr>
<td>Difference</td>
<td>0.00178</td>
<td>0.002037</td>
<td>0.31022</td>
<td>-0.55859</td>
<td>0.1284</td>
<td>0.00016 (0.1%)</td>
</tr>
<tr>
<td>Worst Exper.</td>
<td>0.18849</td>
<td>0.188792</td>
<td>3.18175</td>
<td>0.186408</td>
<td>2.76482</td>
<td>0.19204</td>
</tr>
<tr>
<td>Difference</td>
<td>0.04159</td>
<td>0.041678</td>
<td>0.85405</td>
<td>0.03942</td>
<td>0.65315</td>
<td>0.04093 (27%)</td>
</tr>
</tbody>
</table>

### Table 3. Tuple Latency (Seconds) Under **Global** Scheduling Strategy

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>O4</th>
<th>O5</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical</td>
<td>0.147114</td>
<td>0.146114</td>
<td>0.079383</td>
<td>0.14690</td>
<td>0.075269</td>
<td>0.246832</td>
</tr>
<tr>
<td>Best Exper.</td>
<td>0.152936</td>
<td>0.149876</td>
<td>0.083768</td>
<td>0.15308</td>
<td>0.076646</td>
<td>0.257601</td>
</tr>
<tr>
<td>Difference</td>
<td>0.005822</td>
<td>0.003762</td>
<td>0.004385</td>
<td>0.00618</td>
<td>0.001377</td>
<td>0.010769 (4%)</td>
</tr>
<tr>
<td>Worst Exper.</td>
<td>0.169569</td>
<td>0.162786</td>
<td>0.092675</td>
<td>0.169233</td>
<td>0.084885</td>
<td>0.1285188</td>
</tr>
<tr>
<td>Difference</td>
<td>0.022455</td>
<td>0.016672</td>
<td>0.013292</td>
<td>0.02233</td>
<td>0.009616</td>
<td>0.038356 (15.5%)</td>
</tr>
</tbody>
</table>
Modeling Summary

- Our analytical results agree with experimental results.
- Hence analytical results can be used for effectively estimating system characteristics.
- The tuple latency and queue size can be evaluated.
- Can be used for capacity planning!
Scheduling

- System Characteristics
  - Bursty input characteristics of data streams
  - A large number of continuous queries
  - QoS requirements

- Consequently
  - Need to buffer unprocessed or partially processed tuples in the system
  - Introduces long tuple latency
  - Dynamic throughput (bursty as well)
Approaches

- **Scheduling strategies**
  - Minimize memory requirement
  - Minimize tuple latency
  - Maximize throughput

- **MavStream**
  - Path capacity strategy
  - Segment strategy
  - Simplified segment strategy
  - A hybrid strategy

- **STREAM**
  - Chain scheduling

- **Aurora**
  - Box/superbox scheduling
Scheduling Strategies

- **Properties**
  - Achieve maximal performance within the fixed amount of resources; trade off among different performance metrics
  - Be aware of the unexpected overload situations, and take corresponding actions in a timely manner
  - Guarantee predefined QoS requirements for a query if any, with the help of other mechanisms such as load shedding, admission control
  - Be implemented easily, and run efficiently with a low overhead (static vs. dynamic policies)
Path Capacity Strategy

- Chain scheduling minimized memory usage

- We wanted to find a strategy that minimized the overall tuple latency, which is the amount of time of a tuple stayed in a DSMS.
Impact of schedule strategies

Figure 1. A query execution plan

Table 1. Operator properties

<table>
<thead>
<tr>
<th>Operator Id</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>selectivity</td>
<td>0.2</td>
<td>0.2</td>
<td>0.8</td>
</tr>
<tr>
<td>service time</td>
<td>1</td>
<td>1</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 2. Performance (F: FIFO, C: Chain)

<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>Queue Size</th>
<th>Tuple Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>C</td>
<td>F</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3.0</td>
<td>2.4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>2.2</td>
<td>1.6</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2.0</td>
<td>0.8</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1.2</td>
<td>0.6</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
<td>0.4</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Experimental Evaluation - Setup

- Input streams
  - Bursty input streams – 5 self-similar input streams with burst input periods
  - Each stream is a super imposition of 64 or 128 flows
  - Every 20 minutes, increase input rate by a factor and then decrease it by another factor.

- Continuous Queries
  - 16 SPJ CQs with 116 operators
  - Different selectivities

- Alpha machine: 2GB RAM, Dual Processor
Memory Requirement

- Chain
- PC
- Simplified t
- Threshold

Experiment Time (seconds)

Memory (Kbytes)

Experiment Time (seconds)
Throughput

Experiment Time (seconds)

Throughput (#Tuples/sec.)

- Chain
- PC
- Simplified
- Threshold

Experiment Time (seconds)

Throughput (#Tuples/sec.)

- Chain
- PC
- MOS
Discussions

- **Optimization**
  - Push down the operators with lowest selectivity and fastest processing rate
  - Shorten the length of an operator path

- **Starvation**
  - Periodically schedule the path with the oldest tuple
  - Dynamic priority: introduce the age of an tuple and the queue size to the priority.
Load Shedding

- Continuous Query Model – Basis of Proposed QoS Framework.
- Scheduling Strategies – Resource Allocation
- Load Shedding – Guaranteed Resources
  - Handling burstiness of input data streams
  - Physical implementation of shedders
  - System Load Estimation
  - Optimal location of shedder
  - Shedding-load distribution among shedders
- EStream – An Integrated Model for Stream and Event Processing
What is Load Shedding?

Load shedding is defined as the process of gracefully discarding unprocessed or partial processed tuples without violating defined QoS requirements of CQs in the system.

It is a trade-off between tuple latency and accuracy

- Decrease in accuracy => reduced tuple latency and total memory requirement
**Why load Shedding**

**Phenomena:**

*Dynamic, bursty inputs => bursty load => Periods of overload*

- Short of memory
- Short of CPU cycles

**Solution:**

- Discard some tuples during overload periods

**What you gain:**

- Decrease peak memory requirement
- Shorten tuple latency

**What you lose:**

- Accuracy, however, some applications tolerate certain of errors, but can not tolerate long tuple latency.
Load Shedding Mechanisms

- **Goal:** satisfy QoS requirements of all CQs in the system while minimizing the overall errors introduced by shedding load

- **Issues**
  1. **When to shed load and how much to shed**
     - Estimate system load
  2. **Where to shed**
     - Find optimal location of each shedder
     - Place weight (a shedder) = load_saved/errors_introduced
  3. **How to shed**
     - Distribute total load needed to shed among shedders
     - Activate the shedder that has the biggest place weight
Shedders

Four possible locations of a shedder in a DSMS
CASE d requires minimal memory and introduces less overhead.
Experimental Results: Tuple Latency

Tuple Latency Under the PC (Group 1)

Sharma Chakravarthy: DEBS
2008
Experiment Results: Accuracy

Moving Average Under the PC for Group 1

Sharma Chakravarthy: DEBS
2008
Problems and Solutions

- Continuous Query Model – Basis of Proposed QoS Framework.
- Scheduling Strategies – Resource Allocation
- Load Shedding – Guaranteed Resources
- EStream – An Integrated Model for Stream and Event Processing
  - Comparison of Event and Stream Processing Models
  - 3-Stage Model
  - Semantic Windows
  - Stream Modifiers
Need for monitoring in stream processing

• Computation of stream data
• Monitoring on the output of stream computation
Sharma Chakravarthy: DEBS 2008

Environment Data  Traffic Data  Battlefield/Surveillance  Network traffic/routing data

Stream Processing

Complex Event Processing

Notification, Alerts, Rules

MavEStream: Transforming Pervasive Real-time Data Into Actionable Knowledge

Monitoring Agencies  Emergency Services  Command Center

Network traffic/routing data

Command Center with Monitor

Monitoring Agency with Monitor

MavEStream System

MavEStream System
Traffic Monitoring

- Accident Detection:
  - Car becomes immobile
  - **Followed By** Another car in same segment reduces speed >30%
  - Action
    - Notify nearest police about the accident
    - Notify all cars in upstream
    - Notify toll station so waiting cars are not extra tolled.

Event Processing

- Find cars which are in the same location for 2 mins
- **Followed By**
- **Condition**
  - Car in same segment
  - Find cars which reduced speed by >30% in 2 mins

Stream Processing

Sharma Chakravarthy: DEBS
2008
Why Event and Rule Processing?

- Current DSMSs mainly focus on providing similar functionalities provided by DBMSs. Event and Rule processing is not supported.
- Many stream-based applications considered for stream processing need complex event expressions and rule processing.
- Event and rule processing studied extensively in the context of DBMSs cannot handle complex stream processing.
- By synthesizing both models and combining their strengths, the integrated model is better than sum of its parts.
# Related Work - Event Processing

<table>
<thead>
<tr>
<th>Detection Model</th>
<th>Drawbacks</th>
<th>Not Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ode [VLDB 1991]</strong></td>
<td>Finite Automaton (FA)</td>
<td>Duplication of FA for each unique attribute value</td>
</tr>
<tr>
<td><strong>SAMOS [Quaterly Bulletin on Data Engg. 1993]</strong></td>
<td>Petrinets</td>
<td>Duplicate primitive event node created for composite events.</td>
</tr>
<tr>
<td><strong>Snoop (Sentinel) [SIGMOD 1993, DKE 1994]</strong></td>
<td>Event Detection Graph</td>
<td>Node sharing for composite events Rules for condition and action</td>
</tr>
</tbody>
</table>
Most of the systems process event streams, but none of them have integrated stream and event processing models

- **Events on the Edge** (SIGMOD 2005, demo paper)
  - Primitive events are detected over sensor data
  - Complex event capability provided
- **Tiny DB**: (SIGMOD 2003)
  - Events is important in stream sensors which lack power resources.
  - Captures operating system events.
- **Cougar**: (MDM 2001)
  - Mentioned the concept of events in stream processing.
- **StreamBase/Apama**: (Commercial Financial Systems)
  - Detects for events on financial streams.
  - Lacks Query definition capability and only suited for financial applications.
- **DataFlow Networks**: (IASTED 2004)
  - Defines events using hardware approach for temperature monitoring applications.
Integrated model

Stage 4: Rule Processing

Stage 3: Event Processing

Stage 2: Event Generation

Stage 1: Continuous Query Processing

Stream \( i \) - Incoming Streams

Stream \( k \) - Stream Operators

Stream \( l \) - Join Operators

Stream \( q \) - Rules

Buffer

Mask

Notify Buffer

E \( p \) - Event Nodes

G \( j \) - Event Generator

LDET - LED Thread

Sharma Chakravarthy: DEBS
2008
Continuous Event Query (CEQ) Specification in EStream

- Specifying Continuous Queries, Events and Rules in EStream
- CEQ specification supports:
  - Creating Named CQs
  - Creating Primitive Events based on CQs
  - Creating Composite Events based on CQ-based Primitive events and other Composite events.
  - Creating and Modifying Attribute-based Constraint on CQ-based Primitive Events
  - Creating Rules (condition and action)
Continuous Query and Event Creation

- CQ Creation in MavStream
  
  CREATE CQ car stopped

- Named CQ Creation in EStream
  
  CREATE CQ car stopped AS Immobile

- Event Creation in LED
  
  EVENT EImmobile = (any primitive event)

- Event Creation in EStream
  
  CREATE PRIMITIVE EVENT EImmobile ON Immobile
Attribute-based Constraint Specification

- Creating and Modifying Attribute-based Constraints (Masks) on CQ-based Primitive Event

- Creating Masks on CQ-based Primitive Events in EStream

CREATE PRIMITIVE EVENT Elmmobile ON Immobile
MASK Elmmobile.attr\textsubscript{i}, \ldots, Elmmobile.attr\textsubscript{j}

MODIFY MASK ON Elmmobile
NEW MASK Elmmobile.attr\textsubscript{i}, \ldots, Elmmobile.attr\textsubscript{j}
example CEQ for Traffic monitoring

CREATE CQ car stopped AS Immobile
CREATE CQ speed reduced AS Decrease

CREATE PRIMITIVE EVENT EImmobile ON Immobile
CREATE PRIMITIVE EVENT EDecrease ON Decrease
CREATE COMPOSITE EVENT EAccident ON EImmobile SEQUENCE EDecrease

CREATE RULE AccidentNotify ON EAccident
Condition is EImmobile.segment == EDecrease.segment
Action is
- Notify nearest police about the accident
- Notify all cars in upstream
- Notify toll station so waiting cars are not extra tolled.
Integration of event and stream processing models

Issues for detection of stream tuples as events

- Same or different address space issue
- Generation of event objects issue

Traffic monitoring example
Event Generator Interface

Functions

- Convert stream tuple to event Object
- Insert stream attributes into event objects
- Enqueue the event object into a buffer to be consumed by LED.
Single or Multiple Event Buffer

Application

CQn

Event Generator Interface

LED Buffer

EDG

CQ1

Application

Event Generator Interface

LED

Application

Event Generator Interface

CQn

Event Detector

CQ1

Event Generator Interface

Event Detector

LED
Advantages

- Operator gets scheduled with the query
  - Operator thread suspends when there are no tuples to process.
- Notification of output tuples not required
- Schema of the stream output is already known
Constraint based Reduction of Uninteresting Event

**Mask:**
Condition on the attributes of the event node that is checked before event is detected.

Defined on a primitive event node

CREATE EVENT *eventName* ON CQ Name
MASK *MaskCondition*….

*MaskCondition* is defined on event parameters
What is a Mask cont.

E.g., Detect Accident only in the HOV lane (e.g., lane 6)

- CREATE PRIMITIVE EVENT Elmmobile ON Immobile
  Mask ElImmobile.lane='6'

- CREATE PRIMITIVE EVENT EDecrease ON Decrease
  Mask ElImmobile.lane='6'

Where the Mask is to be checked:
Pushed lowest in the computational model
Where should mask be added

Event computational model

Stream computational model

EG1
Lane = 6
Immobile

EG2
Lane = 6
Decrease

EAccident
ElImmobile
EDecrease

Lane = 6

LED Buffer

Event Detector

Composite Event
Primitive Event
Rule
Experiment: Query With and Without Mask

- Queries with mask and without mask
  
  *Issuing speeding ticket for cars if their speed > 70 mph*

```sql
CREATE CQ AUTOMATEDMONITOR AS
SELECT * FROM CarLocStr

CREATE EVENT "SpeedingTicket" ON AUTOMATEDMONITOR
MASK Speed > 70

CREATE RULE "GiveTicket"
CONDITION Speed > 70
ACTION "GiveSpeedingTicket"
```
• Number of Events generated are reduced with masks
• Query performance is improved with masks
Multiple Masks and Events

- Multiple events can be generated from a single query using multiple masks.

\[
\text{EG: } \text{Speed} > 70 \land \text{Lane} = \text{HOV} \quad \text{Speed} > 70 \land \text{Lane} \neq \text{HOV} \quad \text{Dir} = \text{Upstream}
\]

- Notify Upstream police
- HOV Speeding
- Speeding
- Charge toll

AUTOMATED MONITOR

CarLocStr
Stream Modifiers

Modifies streams based on the change between two consecutive states/tuples of the stream

Types of Stream Modifiers:
- Adiff
- Rdiff
- Slope
- WAdiff
- WRDiff
- WSlope

Example;
- Generate events of selling shares when the price of a share increases by more than 20%.
Stream Modifiers

Notations used:

- **Sub-Tup_i{A_1, A_2...A_m}**: Value of attributes A_1 to A_m in the i^{th} tuple.
  - E.g., Sub-Tup_1{shareName, Price} = {RIL, 12.31}

- **State function S_i(A_j)**: The value of the j^{th} attribute in the i^{th} tuple of the stream
  - E.g., S_1(ShareID)=1

<table>
<thead>
<tr>
<th>ShareID</th>
<th>Share Name</th>
<th>Price $</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RIL</td>
<td>12.31</td>
</tr>
<tr>
<td>1</td>
<td>RIL</td>
<td>11.97</td>
</tr>
<tr>
<td>1</td>
<td>RIL</td>
<td>12.05</td>
</tr>
</tbody>
</table>
Stream Modifiers Input and Output

Input:
Operator \((A_1,A_2,\ldots,A_m), P_{\text{pseudo}}, O/N\)
Where \(m \leq n\), \(O\) is old and \(N\) is new.

Output:
Modified \((A_1,A_2,\ldots,A_m)\), Sub-Tup\(_{(O/N)}\)(\(A_{m+1},\ldots,A_n\))
ADiff Stream Modifier

Adiff: Detects the absolute change over two consecutive states of a stream.

Input
- \( ADiff(A_1, A_2, \ldots, A_m) \ O \)
  - \( m \leq n \) which absolute change is to be computed

Output
- Tuple will follow states

\[
ADiff \ (A_1, A_2, A_3, \ldots, A_m), \ O = \left( \frac{(S_{i+1}(A_1) - S_i(A_1))}{S_i(A_1)} \right) \ldots \left( \frac{(S_{i+1}(A_m) - S_i(A_m))}{S_i(A_m)} \right) + \text{Sub} - \text{Tup}_i(A_{m+1}, \ldots, A_n)
\]
Stream Modifiers cont.

Example,
Adiff (Price) O RIL_NYSE

<table>
<thead>
<tr>
<th>ShareID</th>
<th>Share Name</th>
<th>Price $</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RIL</td>
<td>12.41</td>
</tr>
<tr>
<td>1</td>
<td>RIL</td>
<td>11.67</td>
</tr>
<tr>
<td>1</td>
<td>RIL</td>
<td>12.05</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{ShareID} & \quad \text{Share Name} & \quad \text{Price} \\
1 & \quad \text{RIL} & \quad 12.41 \\
1 & \quad \text{RIL} & \quad 11.67 \\
1 & \quad \text{RIL} & \quad 12.05 \\
\end{align*}
\]

\[
\frac{(11.97-12.41)}{12.41}
\]

\[
\frac{(12.05 - 11.67)}{11.67}
\]

<table>
<thead>
<tr>
<th>ShareID</th>
<th>Share Name</th>
<th>Price Change $</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RIL</td>
<td>-.035</td>
</tr>
<tr>
<td>1</td>
<td>RIL</td>
<td>.032</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{ShareID} & \quad \text{Share Name} & \quad \text{Price Change} \\
1 & \quad \text{RIL} & \quad -.035 \\
1 & \quad \text{RIL} & \quad .032 \\
\end{align*}
\]
Windowed Stream Modifiers

Computes the change between the first and the last tuple of the window.

Example: Generate event when a car accelerates continuously till 10 minutes monitored every 2 min.

Requirements
- Can be used at any level in the query tree
- if used at leaf level
  - Window specification may not be the same of the query
Integrated model

Stage 1: Continuous Query Processing

Stage 2: Event Generation

Stage 3: Event Processing

Stage 4: Rule Processing

Stream \( i \): Incoming Streams

Stream \( S_k \): Stream Operators

Stream \( J_l \): Join Operators

Rule \( R_q \): Rules

Buffer

Mask

Notify Buffer

Event Nodes \( E_p \)

Event Generator \( G_j \)

LEDT - LED Thread

CQ Processing
EStream Architecture

- User Input
- Input Processor
- Rule and Event Manager
- Instantiator
- Scheduler
- Run-Time Optimizer
- Query Processor
- Data Streams
- Feeder
- LED
- Event Detector
- LED Buffer
- EStream Server
Conclusions

- There is considerable amount of work done for stream processing
  - Operator/query modeling
  - Summarization
  - Scheduling
  - Load shedding
- There is considerable amount of work done in CEP
  - Primitive events
  - Composite/complex events
  - Event semantics
  - Implementation
Future trends

- Most of the stream processing is for monitoring purposes
- Hence, results in triggers
- Although there has been significant work in event processing, they were done prior to stream processing
- New needs have to be taken into account
  - Very large and bursty events
  - Best effort may not be sufficient
  - QoS needed for the event part to provide end-to-end tuple latency, memory usage and throughput
  - Scheduling, load shedding for event processing as well
Future trends (2)

- Surprisingly, there is no benchmark for performance
- None of the systems explain how the performance is achieved and what are the issues
- The non-procedural aspects has been somewhat replaced by procedural implementation
- IDE’s make it easy to drag and drop
- Eventually, understanding and maintaining the code are likely to come back and haunt us (once again!)
References


References (Contd.)


References (Contd.)

