Active Database Systems

- An integrated facility for creating and executing production rules from within a database system

- A typical database production rule:
  
  ```
  when event
  if condition
  then action
  ```

- Powerful and uniform mechanism for:
  
  - Constraint enforcement
  - Derived data maintenance
  - Alerting
  - Authorization checking
  - Version management
  - Resource management
  - Knowledge management
Outline of Slides

- Chapter 2: Syntax and Semantics
  - A Relational Prototype: Starburst
  - Two Relational Systems: Oracle and DB2
  - An Object-Oriented Prototype: Chimera
  - Features and Products Overview

- Chapter 3: Applications
  - Applications of Active Rules
  - Deriving Active Rules for Constraint Management
  - Deriving Active Rules for View Maintenance
  - Rules for Replication
  - Rules for Workflow Management
  - Business Rules

- Chapter 4: Design Principles
  - Properties of Active Rules and Rule Analysis
  - Rule Modularization
  - Rule Debugging and Monitoring
  - Rule Design: IDEA Methodology (pointer)
  - Conclusions
A Relational Example: Starburst

- Done at IBM Almaden
- Chief Engineer: Jennifer Widom
- Syntax based on SQL
- Semantics is set-oriented
  - Triggering based on (arbitrary) sets of changes
  - Actions perform (arbitrary) sets of changes
  - Conditions and actions can refer to sets of changes
- Instructions: create, drop, alter, deactivate, activate
Rule Creation

\[<\text{Starburst-rule}> ::= \text{CREATE RULE} \ <\text{rule-name}>\]
\[\quad \text{ON} \ <\text{table-name}>\]
\[\quad \text{WHEN} \ <\text{triggering-operations}>\]
\[\quad \quad [\text{IF} \ <\text{SQL-predicate}> ]\]
\[\quad \text{THEN} \ <\text{SQL-statements}>\]
\[\quad \quad [\text{PRECEDES} \ <\text{rule-names}> ]\]
\[\quad \quad [\text{FOLLOWS} \ <\text{rule-names}> ]\]

\[<\text{triggering-operation}> ::= \text{INSERTED} \mid \text{DELETED} \mid \]
\[\quad \text{UPDATED} [ ( <\text{column-names}> ) ]\]

- Triggering operations:
  - inserted, deleted, updated, updated\((c_1,\ldots,c_n)\)
- Condition: arbitrary SQL predicate
- Actions: any database operations
  - insert, delete, update, select, rollback, create table, etc.
- Precedes and Follows: for rule ordering
Example Rules

Salary control rule
CREATE RULE SalaryControl ON Emp
WHEN INSERTED, DELETED, UPDATED (Sal)
IF (SELECT AVG (Sal) FROM Emp ) > 100
THEN UPDATE Emp
    SET Sal = .9 * Sal

High paid rule
CREATE RULE HighPaid ON Emp
WHEN INSERTED
IF EXISTS (SELECT * FROM INSERTED
    WHERE Sal > 100)
THEN INSERT INTO HighPaidEmp
    (SELECT * FROM INSERTED
    WHERE Sal > 100)
FOLLOWS AvgSal
Transition Tables

- Logical tables storing changes that triggered rule
- Can appear anywhere in condition and action
- References restricted to triggering operations:
  - inserted
  - deleted
  - new-updated
  - old-updated
**Rule Execution Semantics**

- Rules processed at commit point of each transaction
- Transaction’s changes are initial triggering transition
- Rules create additional transitions which may trigger other rules or themselves
- Each rule looks at set of changes since last considered
- When multiple rules triggered, pick one based on partial ordering
Example of rule execution

- Initial state:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stefano</td>
<td>90</td>
</tr>
<tr>
<td>Patrick</td>
<td>90</td>
</tr>
<tr>
<td>Michael</td>
<td>110</td>
</tr>
</tbody>
</table>

- Transaction inserts tuples (Rick, 150) and (John, 120)

<table>
<thead>
<tr>
<th>Employee</th>
<th>Sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stefano</td>
<td>90</td>
</tr>
<tr>
<td>Patrick</td>
<td>90</td>
</tr>
<tr>
<td>Michael</td>
<td>110</td>
</tr>
<tr>
<td>Rick</td>
<td>150</td>
</tr>
<tr>
<td>John</td>
<td>120</td>
</tr>
</tbody>
</table>

- Rule SalaryControl runs:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stefano</td>
<td>81</td>
</tr>
<tr>
<td>Patrick</td>
<td>81</td>
</tr>
<tr>
<td>Michael</td>
<td>99</td>
</tr>
<tr>
<td>Rick</td>
<td>135</td>
</tr>
<tr>
<td>John</td>
<td>108</td>
</tr>
</tbody>
</table>
Rule Execution Semantics (2)

- Rule SalaryControl runs again:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stefano</td>
<td>73</td>
</tr>
<tr>
<td>Patrick</td>
<td>73</td>
</tr>
<tr>
<td>Michael</td>
<td>89</td>
</tr>
<tr>
<td>Rick</td>
<td>121</td>
</tr>
<tr>
<td>John</td>
<td>97</td>
</tr>
</tbody>
</table>

- Rule HighPaid runs eventually, and inserts into HighPaid only one tuple:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rick</td>
<td>122</td>
</tr>
</tbody>
</table>
Oracle

- Supports general-purpose triggers, developed according to preliminary documents on the SQL3 standard.
- Actions contain arbitrary PL/SQL code.
- Two granularities: row-level and statement-level.
- Two types of immediate consideration: before and after.
- Therefore: 4 Combinations:
  
  BEFORE ROW
  BEFORE STATEMENT
  AFTER ROW
  AFTER STATEMENT
### Syntax

\(<\text{Oracle-trigger}> ::= \text{CREATE TRIGGER } <\text{trigger-name}> \\
\quad \{ \text{BEFORE | AFTER} \} <\text{trigger-events}> \\
\quad \text{ON } <\text{table-name}> \\
\quad [ [ \text{REFERENCING } <\text{references}> ] ] \\
\quad \text{FOR EACH ROW} \\
\quad [ \text{WHEN ( } <\text{condition}> \text{ ) } ] ] <\text{PL/SQL block}> \\

\(<\text{trigger event}> ::= \text{INSERT | DELETE | UPDATE} \\
\quad [ \text{OF } <\text{column-names}> ] \\

\(<\text{reference}> ::= \text{OLD AS } <\text{old-value-tuple-name}> | \text{NEW AS } <\text{new-value-tuple-name}> \)
1. Execute the **BEFORE STATEMENT** trigger.

2. For each row affected:
   
   (a) Execute the **BEFORE ROW** trigger.
   
   (b) Lock and change the row.
   
   (c) Perform row-level referential integrity and assertion checking.
   
   (d) Execute the **AFTER ROW** trigger.

3. Perform statement-level referential integrity and assertion checking.

4. Execute the **AFTER STATEMENT** trigger.
Example Trigger in Oracle

Reorder rule
CREATE TRIGGER Reorder
AFTER UPDATE OF PartOnHand ON Inventory
FOR EACH ROW
  DECLARE NUMBER X
  BEGIN
    SELECT COUNT(*) INTO X
    FROM PendingOrders
    WHERE Part = New.Part;
    IF X = 0
      THEN
        INSERT INTO PendingOrders
        VALUES (New.Part, New.OrderQuantity, SYSDATE)
      END IF;
  END;

Example of execution

- Initial state of Inventory:

<table>
<thead>
<tr>
<th>Part</th>
<th>PartOnHand</th>
<th>ReorderPoint</th>
<th>ReorderQuantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200</td>
<td>150</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>780</td>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>450</td>
<td>400</td>
<td>120</td>
</tr>
</tbody>
</table>

- PendingOrders is initially empty

- Transaction (executed on October 10, 1996):

  $T_1$: UPDATE Inventory
  
  
  WHERE Part = 1

- After the execution of trigger Reorder, insertion into PendingOrders of the tuple (1,100,1996-10-10)

- Another transaction (executed on the same day)

  $T_2$: UPDATE Inventory
  
  
  WHERE Part >= 1

- The trigger is executed upon all the tuples, and the condition holds for parts 1 and 3, but a new order is issued for part 3, resulting in the new tuple (3,120,1996-10-10).
• Triggers for DB2 Common Servers defined at the IBM Almaden Research center in 1996.

• Influential on the SQL3 standard.

• As in Oracle: either BEFORE or AFTER their event, and either a row- or a statement-level granularity.

• Syntax:

  \[
  \langle \text{DB2-trigger} \rangle ::= \text{CREATE TRIGGER} \ <\text{trigger-name}> \\
  \{ \ \text{BEFORE} \ | \ \text{AFTER} \ \} \ <\text{trigger-event}> \\
  \text{ON} \ <\text{table-name}> \\
  \ \ [ \ \text{REFERENCING} \ <\text{references}> \ ] \\
  \text{FOR EACH} \ \{ \ \text{ROW} \ | \ \text{STATEMENT} \ \} \\
  \text{WHEN} \ ( \ <\text{SQL-condition}> \ ) \\
  \ <\text{SQL-procedure-statements}>
  \]

  \[
  \langle \text{trigger-event} \rangle ::= \text{INSERT} \ | \ \text{DELETE} \ | \ \text{UPDATE} \\
  \ \ [ \ \text{ON} \ <\text{column-names}> \ ]
  \]

  \[
  \langle \text{reference} \rangle ::= \text{OLD AS} \ <\text{old-value-tuple-name}> \ | \\
  \ \text{NEW AS} \ <\text{new-value-tuple-name}> \ | \\
  \ \text{OLD_TABLE AS} \ <\text{old-value-table-name}> \ | \\
  \ \text{NEW_TABLE AS} \ <\text{new-value-table-name}>
  \]
Semantics of DB2 Triggers

- Before-triggers:
  - Used to detect error conditions and to condition input values (assign values to NEW transition variables).
  - Read the database state prior to any modification made by the event.
  - Cannot modify the database by using UPDATE, DELETE, and INSERT statements (so they do not recursively activate other triggers).

- Several triggers (with either row- or statement-level granularity) can monitor the same event.

- A system-determined total order takes into account the triggers’ definition time; row- and statement-level triggers are intertwined in the total order.

- General trigger processing algorithm after statement A:

  1. Suspend the execution of A, and save its working storage on a stack.
  2. Compute transition values (OLD and NEW) relative to event E.
  3. Consider and execute all before-triggers relative to event E, possibly changing the NEW transition values.
  4. Apply NEW transition values to the database, thus making the state change associated to event E effective.
5. Consider and execute all after-triggers relative to event E. If any of them contains an action $A_i$ that activates other triggers, then invoke this processing procedure recursively for $A_i$.

6. Pop from the stack the working storage for A and continue its evaluation.

- Revised trigger processing with integrity checking:

4. Apply the NEW transition values to the database, thus making the state change associated to event E effective. For each integrity constraint IC violated by the current state, consider the action $A_j$ that compensates the integrity constraint IC.
   a. Compute the transition values (OLD and NEW) relative to $A_j$.
   b. Execute the before-triggers relative to $A_j$, possibly changing the NEW transition values.
   c. Apply NEW transition values to the database, thus making the state change associated to $A_j$ effective.
   d. Push all after-triggers relative to action $A_j$ into a queue of suspended triggers.

Until a quiescent point is reached where all the integrity constraints violated in the course of the computation are compensated.
Examples of triggers

Supplier rule
CREATE TRIGGER OneSupplier
BEFORE UPDATE OF Supplier ON Part
REFERENCING NEW AS N
FOR EACH ROW
WHEN (N.Supplier IS NULL)
    SIGNAL SQLSTATE '70005'
    ('Cannot change supplier to NULL')

Audit rule
CREATE TRIGGER Audit
AFTER UPDATE ON Parts
REFERENCING OLD_TABLE AS OT
FOR EACH STATEMENT
    INSERT INTO AuditSupplier
      VALUES(USER, CURRENT_DATE,
            (SELECT COUNT(*) FROM OT))