Course Notes on
Active-Database Management

What you will learn in this chapter

• an introduction to active-database technology (i.e., triggers), which is a useful extension to classical DBMS technology
• the basic idea is intuitively simple and natural
• the detail of trigger definition gets quickly complicated, especially as triggers are part of the database and must be valid for all application programs
• the practical usage of the technology is mostly about automating simple repetitive operations that would be very time-consuming to be coded in application programs

Active Databases: Topics

• Introduction
• Representative Systems and Prototypes
• Applications of Active Rules
Active-Database Technology

- **Passive DBMS**: all actions on data result from explicit invocation in application programs (they only do what application programs tell them to do)

- **Active DBMS**: execution of actions can be automatically triggered in response to monitored *events*, including
  - database updates (upon deletion of the data about a customer)
  - points in time (on January 1, every hour)
  - events external to the database (whenever paper jams in the printer)

Evolution of database technology has been going thru representing and supporting more functionality of database applications within the DBMS, e.g.,

- checks of some types of **integrity constraints** (produced from a declarative definition located with the database schema)
- **stored procedures**: precompiled procedures located within the database, invoked from application and system programs
- **common semantics** abstracted from application domains (e.g., for spatial, multimedia, temporal, deductive, active databases)

- **Active-database technology**
  - a relatively recent extension of traditional DBMS technology
  - most commercial RDBMSs include some capability for rules or **triggers**
  - research prototypes provide more comprehensive support for active rules than RDBMSs

- Application semantics in programs for active DBMSs is expressed in:
  - traditional application programs (as for passive DBMSs)
  - rules (in the database, available to all applications)
**Event - Condition - Action Rules**

- When an event occurs, if a condition holds, then an action is performed

<table>
<thead>
<tr>
<th>Event</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>a customer has not paid 3 invoices at the due date</td>
<td>if the credit limit of the customer is less than 20 000 Euros</td>
<td>cancel all current orders of the customer</td>
</tr>
</tbody>
</table>

- ECA rules are part of the database (⇒ “rule base”), available to all applications

---

**Rules May Express Various Aspects of Application Semantics**

- **Static constraints** (e.g., referential integrity, cardinality, value restrictions)
  - only regular students can register at the library
  - students can register in no more than 20 courses
  - the salary of employees cannot exceed the salary of their manager

- **Control, business rules, workflow management**
  - when data for new students is recorded, data is automatically entered to register the students in the mandatory courses
  - all expenses exceeding 50K must be approved by a manager
  - when an order has been accepted, an invoice is sent

- **Historical data**
  - the data about completed orders is transferred monthly to the data warehouse
Semantics Modeled by Rules (cont’d)

- Implementation of generic relationships (e.g., generalization)
  - a person is a student or a lecturer, but not both

- Derived data: materialized attributes, materialized views, replicated data
  - the number of students registered in a course must be part of the course data
  - orders received are summarized daily in the planning database

- Access control
  - employees can view data about their own department only

- Monitoring: performance, resource-usage monitoring
  - the number of disk accesses of each database query is recorded and statistics are produced weekly
  - each access to our web pages is reflected in the usage database

- Exercise: rephrase the above examples as event-condition-action rules
- Note that many examples have a more declarative form than ECA rules
Benefits of Active Technology

- **Simplification of application programs**: part of the functionality can be programmed with rules that belong to the database
- **Increased automation**: actions are triggered without direct user intervention
- **Higher reliability** of data thru more elaborate checks and repair actions ⇒ better computer-aided decisions for operational management
- **Increased flexibility** thru centralisation and code reuse ⇒ reduced development and maintenance costs

Active Databases: Topics

- Introduction
- **Representative Systems and Prototypes**
  - Starburst
  - Oracle
  - DB2
  - SQL Server
- Applications of Active Rules
• The basic ECA model for rules is simple and intuitive, but there are significant differences in the ways that they have been realized in practice

• Triggers were not included in the SQL-92 standard

• Starburst triggers are presented for their simplicity

• Oracle triggers are based on an early version (≈ 1993) of the SQL3 standard

• DB2 triggers are closer to more recent SQL3 proposals

---

**Starburst**

• Relational prototype by IBM Almaden Research Center

• Event-Condition-Action rules in Starburst:
  ◇ **event**: data-manipulation operations (INSERT, DELETE, UPDATE) in SQL
  ◇ **condition**: Boolean predicate in SQL on the current state of the database
  ◇ **action**: SQL statements, rule-manipulation statements, rollback
Starburst: Example of Rule Definition

The salary of employees is not larger than the salary of the manager of their department

```
CREATE RULE Mgrsals ON Emp
    WHEN INSERTED, UPDATED(Dno), UPDATED(Sal)
    IF EXISTS (SELECT *
                FROM Emp E, Dept D, Emp M
                WHERE E.Dno = D.Dno AND E.Sal > M.Sal AND D.Mgr = M.Name
               )
    THEN ROLLBACK
```

- Relations $\text{Emp}(\text{Name,Sal,Dno})$ and $\text{Dept}(\text{Dno,Mgr})$
- The rule is activated when an employee is created, and when the salary or the department of an employee changes
- $\text{rollback}$ = abort the transaction with the statement that caused the triggering event
- Remark about the example:
  - The constraint (the salary of employees is not larger than the salary of the manager of their department) is declarative
  - rule $\text{Mgrsals}$ does not cover all the scenarios that could lead to violating the constraint
  - another rule is necessary on relation $\text{Dept}$, to check the constraint when a department changes its manager
Starburst: Syntax of Rule Definition

```
<Starburst-rule> ::= CREATE RULE <rule-name> ON <relation-name>
WHEN <list of trigger-events>
[ IF <condition> ]
THEN <list of SQL-statements>
[ PRECEDES <list of rule-names> ]
[ follows <list of rule-names> ]
```

```
<trigger-event> ::= INSERTED | DELETED | UPDATED [ ( <attributes> ) ]
```

- Each rule has a unique name
- Each rule is associated with a single relation
- Events can only be database updates
- A rule can monitor several events on the target relation
- Events can be monitored by several rules
- The condition is an SQL predicate (true if the result is nonempty)
- Rules are ordered for execution priority through an (acyclic) partial order (PRECEDES, FOLLOWS)
Starburst: Other Example of Rule Definition

If the average salary of employees gets over 100, then reduce the salary of all employees by 10%

CREATE RULE SalaryControl ON Emp
WHEN INSERTED, DELETED, UPDATED(Sal)
IF (SELECT AVG(Sal) FROM Emp) > 100 )
THEN
  UPDATE Emp
  SET Sal = 0.9 * Sal

Example of invocation

INSERT INTO Emp VALUES('Dupont', 90)

---

Starburst: Semantics

- Rules are triggered by the execution of operations in statements that are part of transactions
- Rules are statement-level: they are executed once per statement even for statements that trigger events on several tuples
- Execution mode is deferred:
  - all rules triggered during transaction execution are placed in a conflict set
  - rules are not considered until the end of the transaction (transaction commit) unless an explicit PROCESS RULES is executed in the transaction
- **Transaction** = a piece of program, a sequence of statements, that is to be treated as atomic (i.e., as an indivisible “all-or-nothing” whole) unit of work for some aspect of processing
  - ◦ for execution (example: a transfer of funds for a bank is a composite operation that must be executed entirely or not at all)
  - ◦ for checking constraint satisfaction (constraints may be temporarily violated while the transaction executes, they must be satisfied before and after the transaction)
  - ◦ for concurrency management (the detail of a transaction should not be visible by and should not be interfered with other transactions)
  - ◦ for active-rule execution (here)

- **Statement**: a part of a transaction; a transaction is a sequence of several statements; a statement expresses an operation on the database (here, for active rules, operations are modifications of database tuples)

- **Event**: the occurrence of executing a statement, i.e., a request for executing an operation on the database; the operation is not necessarily executed at the same time that the event occurs

- With the deferred mode, condition evaluation and action execution are decoupled from rule activation

- Vocabulary: *set-oriented* is also used for *statement-level*

---

**Rule Processing**

- **Algorithm for rule selection and execution**

  While the conflict set is not empty
  
  (1) Select a rule \( R \) in the conflict set among those rules at highest priority; take \( R \) out of the conflict set
  
  (2) Evaluate the condition of \( R \)
  
  (3) If the condition of \( R \) is true, then execute the action of \( R \)
Important Moments in Rule Processing

- Although \textit{triggering}, \textit{firing}, or \textit{executing} are often used in practice, processing of a rule involves 3 stages

1. \textbf{Activation} or \textbf{instanciation}: the event in the rule requests the execution of an operation and this is detected by the rule-processing system

2. \textbf{Consideration}: the condition in the rule is evaluated

3. \textbf{Execution}: the action in the rule is executed
Correctness ⇒ Repeatability + Termination

- **Repeatability of execution** = the system behaves in the same way when it receives the same input transaction in the same database state (determinism)
  - rule definitions specify a partial order for execution
  - upon rule selection, several rules may have highest priority at step (1)
  - for repeatability, the system maintains a total order based on user-defined partial order and rule creation time

- **Termination of rule execution** = an empty conflict set is reached
  - action execution may cause repeated rule triggering
  - nontermination: some rules can cyclicly trigger each other
  - ensuring termination: one of the main problems of active-rule design (practical systems impose a limit on the depth of cyclic invocation to ensure termination)

Starburst: Example of Termination

<table>
<thead>
<tr>
<th>Name</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>

CREATE RULE SalaryControl ON Emp
WHEN INSERTED, DELETED, UPDATED(Sal)
IF (SELECT AVG(Sal) FROM Emp) > 100
THEN UPDATE Emp
SET Sal = 0.9 * Sal

- Transaction: insert \{(Rick,150), (John,120)\} into Emp

<table>
<thead>
<tr>
<th>Name</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>
• Here, the action part of the rule causes an event that activates the rule again, in a cascading mode.

<table>
<thead>
<tr>
<th>Name</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>

• Insertion triggers the rule, condition is true (average = 112) ⇒ action executed

<table>
<thead>
<tr>
<th>Name</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>

• Updating triggers the rule again, condition is true (average = 101) ⇒ action executed

<table>
<thead>
<tr>
<th>Name</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>

• The rule is triggered again, the condition is false (average = 91) ⇒ termination

• The action of SalaryControl updates all tuples of Emp, but these updates lead to a single consideration of SalaryControl (statement-level semantics)

• Termination for the example:
  ◦ rule processing terminates because the multiplicative factor for salaries is 1: the action of the rule decreases the average salary and the condition of the rule checks that the average salary is below a given value ⇒ eventually the average salary will go below the given value
  ◦ if the multiplicative factor of the salaries was < 1, then rule application would not terminate

• In general guaranteeing termination
  ◦ is the responsibility of the programmer
  ◦ is not easy
State Transitions and Net Effect (1)

• A transaction causes a state transition of the database, by adding, suppressing, and/or modifying database tuples

• INSERTED, DELETED: temporary transition relations containing the tuples expressing the transition

• Net effect: compose multiple operations occurring on the same tuple
  ◦ a sequence of INSERT and DELETE on the same tuple, with any number of intermediate UPDATE, has a null effect
  ◦ an INSERT and several UPDATE on the same tuple are equivalent to a single INSERT
  ◦ several UPDATE and a DELETE on the same tuple are equivalent to a DELETE

State Transitions and Net Effect (2)

• Transition relations at the end of execution (with net effect)

<table>
<thead>
<tr>
<th>INSERTED</th>
<th>DELETED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Sal</td>
</tr>
<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>

• Rules consider the net effect of operations between two database states ⇒ each tuple appears at most once in each temporary table
Precise Definition of RuleTriggering

- A rule is triggered if any of the transition relations corresponding to its triggering operations is not empty.
- Rule can reference transition relations (this can be more efficient than referring to database relations).

Rule evaluation is governed by the net effect of operations: transition relations contain the net effect of all operations within the transaction that cause transitions between database states.

- multiple operations occurring on the same tuple are composed (e.g., insertion of a tuple followed by deletion of the same tuple has no net effect) \( \Rightarrow \) each tuple appears at most once in each of the sets INSERTED and DELETED.
- More than one of INSERTED and DELETED can be nonempty for the same rule execution as a rule can monitor several events.
Starburst: Example of Rule Priority and Net Effect

- Add a new rule `HighPaid` in addition to rule `SalaryControl`

```
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 SalaryControl
```

- Transaction: insert `{(Rick, 150), (John, 120)}` into `Emp`
- `SalaryControl` and `HighPaid` are both activated
- `SalaryControl` first executes twice, yielding the same state as above for `Emp`
- `HighPaid` then executes and inserts `(Rick, 121)` into `HighPaidEmp`

- Rule `HighPaid` inserts highly paid employees into relation `HighPaidEmp`
- Insertion of `(Rick, 150)` and `(John, 120)` triggers both `SalaryControl` and `HighPaid`
- `INSERTED(Emp) = {(Rick, 121), (John, 97)}` when the condition of `HighPaid` is evaluated and its action executed
- Thus `HighPaid` is triggered and executed in different database states
  - it is triggered at the same time as `SalaryControl` (at the first insertion into `Emp`)
  - it is considered and executed after all executions of `SalaryControl`
- Note that, by the application of semantics `HighPaid`, it is logical that it be executed after the adjustments made by `SalaryControl`
- Exercises:
  - add `Updated(Sal)` to the triggering events of `HighPaid`
  - discuss the behavior of the rules when `FOLLOWS SalaryControl` is removed from rule `HighPaid`
### Starburst: Other Commands

- `<deactivate-rule>` ::= DEACTIVATE RULE `<rule-name>` ON `<table-name>`
- `<activate-rule>` ::= ACTIVATE RULE `<rule-name>` ON `<table-name>`
- `<drop-rule>` ::= DROP RULE `<rule-name>` ON `<table-name>`
- `<create-ruleset>` ::= CREATE RULESET `<ruleset-name>`
- `<alter-ruleset>` ::= ALTER RULESET `<ruleset-name>`
  - `[ ADDRULES `<rule-names>` ]`
  - `[ DELRULES `<rule-names>` ]`
- `<drop-ruleset>` ::= DROP RULESET `<ruleset-name>`
- `<rule-processing-commands>` ::= PROCESS RULES
  - | PROCESS RULESET `<ruleset-name>`
  - | PROCESS RULE `<rule-name>`

- Transactions can dynamically activate and deactivate existing rules
- Rules may be grouped in sets
- Rule processing can be started from within a transaction thru `PROCESS RULE` to counter the default deferred mode (whereby rules are executed at the commit time of the transaction)
- Rule processing can be required to concern a specific rule, a specific set of rules, or all rules
Oracle: Triggers

- Respond to modification operations (insert, delete, update) to a relation
- Granularities for rules
  - `tuple-level` (or row-level): a rule is triggered once for each tuple concerned by the triggering event
  - `statement-level`: a rule is triggered only once even if several tuples are involved
- Immediate execution mode: rules are considered immediately after the event has been requested (Starburst rules are deferred)
- Rules can be considered and executed before, after, or instead of the operation of the triggering event is executed

Oracle: a Simple Example

```
CREATE TRIGGER NondecSal
AFTER UPDATE OF Sal ON Emp
REREFENCING OLD AS t1
    NEW as t2
FOR EACH ROW
WHEN (t2.SAL > t1.SAL)
    UPDATE Emp
    SET Sal = t1.Sal
    WHERE Name = t2.Name
```
The relation concerned is \textbf{Emp(Name,Sal,Dno)}

\textbf{NondecSal} is the trigger name.

t1 and t2 are like tuple variables in SQL: they serve to refer, in the condition and action parts, to the old tuple (before the update) and the new tuple (after the update).

\textbf{FOR EACH ROW} expresses the requirement that the trigger is executed once for each updated tuple.

---

**Oracle: Syntax of Trigger Definition**

```
<Oracle-trigger> ::= CREATE TRIGGER <trigger-name>
   { BEFORE | AFTER } <list of trigger-events>
   ON <table-name>
   [ [ REFERENCING <references> ]
   FOR EACH ROW [ WHEN ( <condition> ) ] ]
   <actions>

<trigger-event> ::= INSERT | DELETE | UPDATE [ OF <column-names> ]

<references> ::= OLD AS <old-value-tuple-name> |
    NEW AS <new-value-tuple-name>

<actions> ::=<PL/SQL block>
```

---

- Row-level triggers versus statement-level triggers: the real issue is what is considered an “event” (execute a statement versus change a database tuple).

- **Row-level triggers** are requested with a \textbf{FOR EACH ROW} clause:
  - the rule is activated once for each tuple concerned
  - useful if the code in the actions depends on data provided by the triggering statement or on the tuples affected
  - \textbf{INSERTING}, \textbf{DELETING}, \textbf{UPDATING} may be used in the action to check which triggering event has occurred
  - the old and the new value of an affected tuple can be referred to by
    * variables introduced in a \textbf{REFERENCING} clause
    * built-in variables \textbf{OLD} (referring to a deleted tuple or to a tuple before it was updated) and \textbf{NEW} (referring to a newly inserted or newly updated tuple)
    * in case of insertion, only the new state is defined; in case of deletion, only the old state is defined
  - the condition consists of a simple predicate on the current tuple
• **Statement-level triggers**
  ◦ this is the default (i.e., implied when the phrase **FOR EACH ROW** is omitted
  ◦ the rule is activated once for each triggering statement even if several tuples are involved (e.g., an SQL statement that updates several tuples) or if no tuple is updated
  ◦ references for old and new tuples are meaningless
  ◦ useful if the code in the actions does not depend on the data provided by the triggering statement nor on the tuples affected (e.g., some security check on user, some audit based on the type of triggering statement)
  ◦ do not have a condition part (it is not clear why Oracle made that decision)
  ◦ do not have the possibility to refer to intermediate relation value thru INSERTED, DELETED, UPDATED as in Starburst (it is not clear why Oracle made that decision)

• Remember that Starburst has statement-level triggers only

---

**Oracle: Example of Row-Level After Trigger (1)**

```
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.ReorderQty, SYSDATE)
        ENDIF;
    END;
```

---

• A classical warehouse-management problem, with two relations:
  ◦ **Inventory**(Part, PartOnHand, ReorderPoint, ReorderQty)
  ◦ **PendingOrders**(Part, Qty, Date)

• Oracle triggers may execute actions containing arbitrary PL/SQL code (not just SQL as for Starburst); PL/SQL extends SQL by adding the typical constructs of a programming language

• Constraint: there is at most one order per part in **PendingOrders** (Part is the key of **PendingOrders**). The tuple is suppressed from **PendingOrders** when the corresponding parts are supplied to the warehouse.
The Reorder rule generates a new order (i.e., inserts a tuple into PendingOrders) whenever the quantity PartOnHand of a given part in Inventory falls below ReorderPoint.

### Oracle: Example of Row-Level After Trigger (2)

<table>
<thead>
<tr>
<th>Part</th>
<th>PartOnHand</th>
<th>ReorderPoint</th>
<th>ReorderQty</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>

- Transaction executed on October 10, 2000
  
  ```sql
  UPDATE Inventory
  WHERE Part = 1
  ```

  - Result: insertion of (1,100,2000-10-10) into PendingOrders

- Transaction executed the same day
  
  ```sql
  UPDATE Inventory
  WHERE Part >= 1
  ```

  - Result: insertion of (3,120,2000-10-10) into PendingOrders

### Exercises:

- write this example for Starburst
- adapt rule Reorder to take into account updates of ReorderPoint and ReorderQty
Oracle: Rule Processing Algorithm

1. Execute the statement-level before-triggers
2. For each row affected by the triggering statement
   (a) Execute the row-level before-triggers
   (b) Execute the modification of the row, check row-level constraints and assertions
   (c) Execute the row-level after-triggers
3. Perform statement-level constraint and assertion checking
4. Execute the statement-level after-triggers

• The execution of data-management statements (insert, delete, or update) in SQL is interwoven with the execution of triggers that are activated by them, according to the preceding algorithm

• Two granularities and two triggering times ⇒ 4 types of triggers:
  ◦ row-level before-triggers (fired before modifying each tuple affected by the triggering statement)
  ◦ statement-level before-triggers (fired once before executing the triggering statement)
  ◦ row-level after-triggers (fired after modifying each row affected by the triggering statement)
  ◦ statement-level after-triggers (fired once after executing the triggering statement)

• The execution also takes into account the checking of constraints and assertions

• Priority among triggers of the same type (row/statement + before/after)
  ◦ early versions of Oracle forbade more than one trigger of the same type
  ◦ more recent versions relaxed the restriction, but do not allow to specify priorities among triggers activated by the same event and with the same type: the order is controlled by the system
Oracle: Semantics of Triggers

- The action part may activate other triggers: the execution of the current trigger is suspended and the others are considered using the algorithm above
  - the maximum number of cascading (i.e., active) triggers is 32
  - when the maximum is reached, execution is suspended and an exception is raised
- If an exception is raised or an error occurs
  - the changes made by the triggering statement and the actions of triggers are rolled back
  - Oracle thus supports partial (per statement) rollback instead of transaction rollback

Instead-of Triggers

- Another mode of specifying triggers, besides before and after triggers
- The action of the instead-of trigger is carried out in place of the statement that produced the activating event
- Instead-of triggers are typically used to update views
- Their power must be contolled (allowing “do X instead of Y” in general would lead to complex effects)
Example of Instead-of Trigger

CREATE TRIGGER manager-insert
INSTEAD OF INSERT ON Managers
REFERENCING NEW AS n
FOR EACH ROW
  UPDATE Dept d
  SET mgrno = n.empno
  WHERE d.deptno = n.deptno

• Relations
  ◦ Emp(empno,empname,deptno)
  ◦ Dept(deptno,deptname,mgrno)

• View:

  CREATE VIEW Managers AS
  SELECT d.deptno, d.deptname, e.empno, e.empname
  FROM Emp e, Dept d
  WHERE e.empno = d.mgrno

• An insert into Managers is interpreted as an update of the mgrno attribute of the corresponding dept tuple
DB2: Syntax of Trigger Definition

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

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

\[\text{<references>} ::= \text{OLD AS} <\text{old-value-tuple-name}> \mid \text{NEW AS} <\text{new-value-tuple-name}> \mid \text{OLD_TABLE AS} <\text{old-value-table-name}> \mid \text{NEW_TABLE AS} <\text{new-value-table-name}> \]

- Every trigger monitors a single event (≠ Starburst, Oracle, Chimera)
- Triggers are activated immediately, BEFORE or AFTER their event, have row- or statement-level granularity, as in Oracle
- State-transition values are defined for both row- and statement-level granularities
  - OLD and NEW introduce tuple values (as in Oracle)
  - OLD_TABLE and NEW_TABLE for relations (as in Starburst)
    * insertion is described by NEW_TABLE only, deletion by OLD_TABLE only
    * OLD_TABLE and NEW_TABLE are equivalent to the temporary relations INSERTED and Deleted in Starburst (remember that Starburst rules can monitor multiple events)
- Triggers cannot execute data definition or transactional commands
- They can raise errors which in turn can cause statement-level rollbacks
DB2: Semantics of Triggers (1)

- Before-triggers
  - Used to detect error conditions and to condition input values
  - Executed entirely before the associated event: their conditions and actions access the database state before any modification made by the event
  - Cannot modify the DB so that they do not recursively activate other triggers

- After triggers
  - Embed part of the application logic in the DB
  - Condition evaluated and action possibly executed after event’s modification
  - State of the DB prior to the event can be reconstructed from transition values
  - E.g., before state of a target relation $T$
    $\left( T \setminus \text{NEW\_TABLE} \right) \cup \text{OLD\_TABLE}$

DB2: Semantics of Triggers (2)

- Several triggers can monitor the same event
- Considered according to a system-defined total order based on creation time of triggers
- Row-level and statement-level triggers intertwined in the total order
- Row-level triggers: considered and possibly executed once for each tuple
- Statement-level triggers: considered and possibly executed once per statement
- If the action of a row-level trigger has several statements, they are all executed for one tuple before considering the next one
DB2: Statement Processing Procedure (1)

- When triggers activate each other: if a modification statement $S$ in the action $A$ of a trigger causes event $E$

1. Suspend the execution of $A$, save its working storage on a stack
2. Compute transition values (OLD and NEW) relative to $E$
3. Consider and execute all before-triggers relative to $E$, possibly changing NEW transition values
4. Apply NEW transition values to DB, making effective state change associated to $E$
5. Consider and execute all after-triggers relative to $E$. If any of them contains an action $A_i$ activating other triggers, invoke this procedure recursively
6. Steps (1) and (6) are not required when $S$ is part of a user transaction
- If an error occurs during the chain processing of $S \Rightarrow$ the prior DB state is restored

DB2: Statement Processing Procedure (2)

- Procedure modified when at (4) $S$ violates SQL-92 constraints: referential constraints, check constraints, views with check option
- After (4) the compensating actions invoked by these constraints are performed until a fixpoint is reached where all ICs are satisfied
- Computations of these actions may activate before- and after-triggers
- Revised Step (4)

4. Apply NEW transition values to DB making effective state change associated to $E$. For each IC violated by current state with compensating action $A_j$
   a. Compute transition values OLD and NEW relative to $A_j$
   b. Execute before triggers relative to $A_j$, possibly changing NEW transition values
   c. Apply NEW transition values to DB making effective state change associated to $A_j$
   d. Push all after-triggers relative to action $A_j$ into a queue of suspended triggers
A referential-integrity constraint requires Part Suppliers to be also distributors, with HDD as a default Supplier.

```
FOREIGN KEY (Supplier)
  REFERENCES Distributor (Name)
  ON DELETE SET DEFAULT
```

- Trigger forcing a statement-level rollback when updating Supplier to NULL

```
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')
```

- Relations Part(PartNum,Supplier,Cost), Distributor(Name,City,State), and Audit(User,CurrDate,UpdTuples): with parts and their suppliers, distributors who are also suppliers, and an audit of updates to the Supplier relation
DB2: Example of Trigger (2)

- Trigger fired at each update of Supplier

  ```
  CREATE TRIGGER AuditSupplier
  AFTER UPDATE ON Part
  REFERENCING OLD_TABLE AS OT
  FOR EACH STATEMENT
  INSERT INTO Audit
  VALUES (User, CurrDate, (SELECT COUNT(*) FROM OT))
  ```

- Transaction executed by Bill on October 10, 1997

  ```
  DELETE FROM Distributor
  WHERE State = 'CA'
  ```

<table>
<thead>
<tr>
<th>Audit</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>Bill</td>
</tr>
</tbody>
</table>

SQL Server: Syntax of Trigger Definition

```
<SQLServer-trigger> ::= CREATE TRIGGER <trigger-name>
ON { <table> | <view> }
[WITH ENCRYPTION]
{ FOR | AFTER | INSTEAD OF }
<list of trigger-events>
[WITH APPEND]
[NOT FOR REPLICATION]
AS
<Transact-SQL-statements>

<trigger-event> ::= INSERT | DELETE | UPDATE
```

SQL Server Triggers

- A single trigger can run multiple actions, it can be fired by more than one event
- Triggers can be attached to tables and views
- Two classes of triggers
  - **INSTEAD OF** triggers: bypass the triggering action and run in their place
  - **AFTER** triggers: fire as a supplement to the triggering action
- Triggers fired with `INSERT`, `UPDATE`, and `DELETE` statements
- **WITH ENCRYPTION**: encrypts the text of the trigger in the `syscomments` table
- **FOR** clause: synonymous with the **AFTER** clause
- **WITH APPEND** and **FOR** used for backward compatibility, not supported in the future
- **NOT FOR REPLICATION**: states that the trigger must not be executed when a replication process modifies the table to which the trigger is attached

SQL Server: INSTEAD OF vs AFTER Triggers

- **INSTEAD OF** triggers
  - Defined on a table or a view
  - Triggers on a view extend the types of updates that a view can support
  - Only one per triggering action is allowed on a table or view
  - Views can be defined on other views, each having its own **INSTEAD OF** trigger
- **AFTER** triggers
  - Defined on a table
  - Modifications to views will fire **AFTER** triggers when table data is modified in response to the view modification
  - More than one allowed on a table
  - Order of execution: Can define which trigger fires first and last using the `sp_settriggerorder` system stored procedure. All other triggers applied to a table execute in random order
SQL Server: Semantics of Triggers

- Both classes of triggers can be applied to a table
- If both trigger classes and constraints are defined for a table, the INSTEAD OF trigger fires
- Then, constraints are processed and AFTER triggers fire
- If constraints are violated, INSTEAD OF trigger actions are rolled back
- AFTER triggers do not execute if constraints are violated or if some other event causes the table modification to fail
- Like stored procedures, triggers can be nested up to 32 levels deep and can be fired recursively

SQL Server: Trigger Actions (1)

- Two transition tables are available: Inserted and Deleted
  - When an insert fires a trigger, the new data is stored in table Inserted
  - When an update fires a trigger, the old data is stored in table Deleted and the new data in table Inserted
  - When a delete fires a trigger, the deleted data is stored in table Deleted
- IF UPDATE(<column-name>) clause: determines whether an INSERT or UPDATE event occurred to the column
  
```sql
IF UPDATE (first_name) OR UPDATE (Last_Name)
BEGIN
  <SQL-statements>
END
```
- COLUMNS_UPDATED() clause returns a bit pattern indicating which of the tested columns were inserted or updated
**SQL Server: Trigger Actions (2)**

- **@@ROWCOUNT**: function returning the number of rows affected by the previous Transact-SQL statement in the trigger
- A trigger still fires event if no rows are affected by the triggering event
- Therefore, the RETURN system command used to exit the trigger transparently when no table modifications are made
- **RAISERROR** system command: used to display error messages
- Transact-SQL statements that are not allowed in a trigger
  - ALTER, CREATE, DROP, RESTORE, and LOAD DATABASE
  - LOAD and RESTORE LOG
  - DISK RESIZE and DISK INIT
  - RECONFIGURE
- If in a trigger’s code it is needed to assign variables SET NOCOUNT ON must be included in the trigger code, disallowing the messages stating how many tuples were modified in each operation

**SQL Server Triggers: Limitations**

- **INSTEAD OF DELETE** and **INSTEAD OF UPDATE** triggers cannot be defined on tables that have corresponding ON DELETE or ON UPDATE cascading referential integrity defined
- Triggers cannot be created on a temporary or system table but the Transact-SQL language within the trigger can reference temporary tables or system tables
Nested and Recursive Triggers

- **Nested trigger option**: determines whether a trigger can be executed in cascade (executes an action that activates another trigger, and so on): There is a limit of 32 nested trigger operations
  
  `sp_configure 'nested triggers', 1 | 0`

- **Recursive trigger option**: causes triggers to be re-fired when the trigger modifies the same table as it is attached to: requires the nested trigger option is set
  
  `sp_dboption '<db-name>', 'recursive triggers', 'TRUE' | 'FALSE'`

- Recursion may be direct or indirect (e.g., a trigger T1 fires a trigger T2 that fires again trigger T1)
- The recursive trigger option only copes with direct recursion
- Indirect recursion coped with nested trigger option

SQL Server: Trigger Management

- Management tasks include altering, renaming, viewing, dropping, and disabling triggers
- Triggers modified with the `ALTER TRIGGER` statement in which the new definition is provided
- Triggers renamed with the `sp_rename` system stored procedure
  
  `sp_rename @objname = <old-name>, @newname = <new-name>`

- Triggers viewed by querying system tables or by using the `sp_helptrigger` and `sp_helptext` system stored procedures
  
  `sp_helptrigger @tabname = <table-name>`
  `sp_helptext @objname = <trigger-name>`

- Triggers deleted with the `DROP TRIGGER` statement
- Triggers enabled or disabled with the `ENABLE TRIGGER` and `DISABLE TRIGGER` clauses of the `ALTER TABLE` statement
SQL Server: Example Trigger (1)

- A book shop database containing the following tables
  - Books(TitleId, Title, Publisher, PubDate, Edition, Cost, QtySold)
  - Orders(OrderId, CustomerId, Amount, OrderDate)
  - BookOrders(OrderId, TitleId, Qty)

- Books.QtySold is a derived attribute keeping track of how many copies of the book has been sold

- A trigger sets this value upon updates of the BookOrders table

```sql
CREATE TRIGGER update_book_qtysold
ON BookOrders
AFTER INSERT, UPDATE, DELETE
AS
IF EXISTS (SELECT * FROM inserted)
BEGIN
    UPDATE Books
    SET QtySold = QtySold +
        (SELECT sum(Qty) FROM inserted i WHERE titleid = i.titleid )
    WHERE titleid IN (SELECT titleid FROM inserted)
END

IF EXISTS (SELECT * FROM deleted)
BEGIN
    UPDATE Books
    SET QtySold = QtySold -
        (SELECT sum(Qty) FROM deleted d WHERE titleid = d.titleid )
    WHERE titleid IN (SELECT d.titleid FROM deleted d)
END
```
SQL Server: Example Trigger (2)

- When an insert to the BookOrders table occurs the trigger fires and updates the QtySold column for the matching TitleID value of the Books table.
- When a delete in the BookOrders table occurs the trigger fires and updates the QtySold column for the matching TitleID.
- An UPDATE event always creates both the Inserted and Deleted tables.
- Therefore, the first part of the code sets the QtySold for the new TitleID value.
- The second part of the code detects the Deleted table and sets the QtySold for the original TitleID value.

Active Databases: Topics

- Introduction
- Representative Systems and Prototypes

Applications of Active Rules
- Integrity Maintenance
- Management of Derived Data
- Management of Replication
- Business Rules
Internal and External Applications

- **Internal applications**: rules support functions provided by specific subsystems in passive DBMSs
  - management of integrity constraints, of derived data, of replicated data, version maintenance, workflow management
  - rules can often be declaratively specified, generated by the system and hidden to the user

- **External applications** or **business rules** (application-domain knowledge):
  - perform computation traditionally expressed in application code
  - rules = faithful modeling of some systems with naturally reactive behavior; rules respond to external events while pursuing some objective (profit maximization, safety, optimal logistics, ...)
  - most interesting when rules express central policies (knowledge common to applications is encoded in rules shared by all applications)

- Some examples of applications that can benefit from active technology and business rules:
  - monitoring access to a building and reacting to abnormal circumstances
  - watching evolution of share values on stock market and triggering trading actions
  - managing inventory to follow stock variations
  - managing a network for energy distribution (see later)
  - airway assignment in air traffic control

- **Notification** (**alerters**): frequent case of application-specific rules whose actions signal, e.g., thru messages, certain conditions that occur with or without changing the database
  - application inserts in the database regular readings by temperature sensors; active rules monitor exceptional temperature values and raise alarms
A Mini-Tutorial on Integrity Constraints

- **Integrity** = accuracy, adequacy, correctness
  More precisely: consistency, conformity of the data with the database schema (including constraints)

- **Integrity constraint**: any prescription (or assertion) on the schema (i.e., valid for all DB extensions) not defined in the data-structure part of the schema

- Constraints declaratively specify conditions to be satisfied by the data at all times ⇒ checking for integrity violations is done for updates

- ICs can be
  - **static**: predicates evaluated on database states
  - **dynamic**: predicates evaluated on state transitions

Integrity Management

- For DBMSs, constraints can be
  - **built-in**, defined by special DDL constructs (e.g., keys, nonnull attributes, unique attributes, referential integrity in RDBMSs)
  - **adhoc**, arbitrarily complex domain-dependent constraints

- Integrity maintenance in practice
  - DBMSs check built-in constraints with automatically generated triggers (e.g., for referential integrity)
  - DBMSs support limited forms of adhoc constraints (e.g., with some version of assertions and **CHECK** clauses of SQL-92)
  - the remaining constraints are implemented as active rules (or in application programs ...)

Active Database Systems- 37
Rule Generation for Integrity Checking

- Rule generation may partially automated:
  (1) possible causes of violation ⇒ events for rule activation
  (2) declarative formulation of the constraint ⇒ rule condition
  (3) avoid or eliminate violation ⇒ action
    ◦ simple approach: force a transaction rollback (abort rules)
    ◦ richer approach: domain-dependent corrective action (repair rules)

Abort Rules and Repair Rules for Integrity

- Abort rules
  ◦ check that integrity is not violated
  ◦ prevent the execution of the offending operation with a rollback command

- Repair rules are sophisticated than abort rules: repair rules embody application-domain semantics
  ◦ constraint specified with actions for restoring integrity
  ◦ example: referential integrity in SQL-92
    * same events and condition as abort rules
    * actions express database manipulations to correct violations (CASCADE, RESTRICT, SET NULL, SET DEFAULT)
Example: Referential Integrity in Starburst

- Relations: Emp(EmpNo, DeptNo, ...), Dept(DNo, ...)
- Referential integrity: Emp[DeptNo] ⊆ Dept[DNo]
  (the department assigned in Emp to every employee must be found in Dept)
- Possible violations upon
  - insertion into Emp
  - deletion from Dept
  - update of Emp.DeptNo
  - update of Dept.DNo
- Condition on tuples of Emp for not violating the constraint
  \[ \text{EXISTS} \left( \text{SELECT} * \text{ FROM Dept WHERE DNo=} \text{Emp.DeptNo} \right) \]
- Denial form (condition for violating the constraint)
  \[ \text{NOT EXISTS} \left( \text{SELECT} * \text{ FROM Dept WHERE DNo=} \text{Emp.DeptNo} \right) \]

Example: Abort Rules in Starburst

CREATE RULE DeptEmp1 ON Emp
WHEN INSERTED, UPDATED(DeptNo)
IF EXISTS (SELECT * FROM Emp WHERE NOT EXISTS
  (SELECT * FROM Dept WHERE DNo=Emp.DeptNo))
THEN ROLLBACK

CREATE RULE DeptEmp2 ON Dept
WHEN DELETED, UPDATED(DNo)
IF EXISTS (SELECT * FROM Emp WHERE NOT EXISTS
  (SELECT * FROM Dept WHERE DNo=Emp.DeptNo))
THEN ROLLBACK
• One rule is necessary for each relation (Emp and Dept)
• The above rules are inefficient: computation of the condition checks the whole database
• Rules may assume that the constraint is verified in the initial state ⇒ it suffices to compute the condition relative to transition values

Example: Repair Rules in Starburst

• More elaborate rules that use transition values for efficiency
• One rule for each event producing a violation
• Choice of application semantics policy
  ◦ for Emp
    * set the DeptNo of new violating employees to NULL
    * set the DeptNo of modified violating employees to the 99 default
  ◦ for Dept
    * when deleting or modifying a department, delete employees whose department number no longer exists (cascade delete policy)
Repair Rules on Relation Emp

CREATE RULE DeptEmp1 ON Emp
WHEN INSERTED
IF EXISTS (SELECT * FROM INSERTED I WHERE NOT EXISTS
(SELECT * FROM Dept D WHERE D.DNo=I.DeptNo))
THEN UPDATE Emp
    SET DeptNo = NULL
    WHERE EmpNo IN (SELECT EmpNo FROM INSERTED I) AND
    NOT EXISTS (SELECT * FROM Dept D WHERE D.DNo=I.DeptNo))

CREATE RULE DeptEmp2 ON Emp
WHEN UPDATED(DeptNo)
IF EXISTS (SELECT * FROM INSERTED I JOIN DELETED D ON
    I.EmpNo=D.EmpNo WHERE NOT EXISTS
    (SELECT * FROM Dept D WHERE D.DNo=I.DeptNo))
THEN UPDATE Emp
    SET DeptNo = 99
    WHERE EmpNo IN (SELECT EmpNo FROM INSERTED JOIN DELETED D
    ON I.EmpNo=D.EmpNo)
    AND NOT EXISTS (SELECT * FROM Dept D WHERE D.DNo=Emp.DeptNo))

Repair Rule on Relation Dept

CREATE RULE DeptEmp3 ON Dept
WHEN UPDATED(DNo), DELETED
IF EXISTS (SELECT * FROM Emp WHERE EXISTS
(SELECT * FROM DELETED D WHERE D.DNo=Emp.DeptNo))
THEN DELETE FROM Emp
    WHERE EXISTS
    (SELECT * FROM DELETED D WHERE D.DNo=Emp.DeptNo))
Management of Derived Data

- **View** = query on the DB returning a relation or a class that can be used as any other relation or class
- **Derived attribute** = whose value can be computed from other DB attributes
- View or attribute derivation is expressed with declarative query language (SQL) or deductive rules (extending query language, e.g., with recursion for transitive closure)
- Two strategies for derived data
  - virtually supported: content computed on demand
  - materialized: content stored in the database ⇒ must be recomputed whenever the source data changes

Virtual Views with Rules

- When an application queries a view
  - a rule is triggered on the request (on SELECT)
  - the action substitutes and evaluates the view definition
- Requires
  - event triggered by queries
  - **INSTEAD OF** clause in rule language
Materialized Views with Rules

- Two basic strategies:
  - \textbf{refresh}: recompute view from scratch after each update of the source data
  - \textbf{incremental}: compute changes to the view from changes in the source relations \Rightarrow positive and negative deltas (tuples to be created or deleted)

- Rule generation can be automated
  - \textbf{refresh} rules are simple but can be inefficient
  - \textbf{incremental} rules depend on the structure of derivation rules and can be complex (e.g., for recursion, duplicates, tuples with several derivations)

- Transform deductive rules into active rules!

Derived Data: Example

\begin{verbatim}
DEFINE VIEW HighPaidDept AS
 ( SELECT DISTINCT Dept.Name
   FROM Dept, Emp
   WHERE Dept.DNo = Emp.DeptNo AND Emp.Sal > 50K )
\end{verbatim}

- Events that may cause the recomputation of the view
  - \textbf{Emp} relation: insertion, deletion, update of \texttt{DeptNo}, update of \texttt{Sal}
  - \textbf{Dept} relation: insertion, deletion, update of \texttt{Dno}
Refresh Rules in Starburst

CREATE RULE RefreshHighPaidDept1 ON Emp
    WHEN INSERTED, DELETED, UPDATED(DeptNo), UPDATED(Sal)
    THEN DELETE * FROM HighPaidDept;
    INSERT INTO HighPaidDept
    ( SELECT DISTINCT Dept.Name
    FROM Dept, Emp
    WHERE Dept.DNo = Emp.DeptNo AND Emp.Sal > 50K )

CREATE RULE RefreshHighPaidDept2 ON Dept
    WHEN INSERTED, DELETED, UPDATED(DNo)
    THEN DELETE * FROM HighPaidDept;
    INSERT INTO HighPaidDept
    ( SELECT DISTINCT Dept.Name
    FROM Dept, Emp
    WHERE Dept.DNo = Emp.DeptNo AND Emp.Sal > 50K )

Incremental Rule for Insert on Dept

CREATE RULE IncRefreshHighPaidDept1 ON Dept
    WHEN INSERTED
    THEN INSERT INTO HighPaidDept
    ( SELECT DISTINCT Dept.Name
    FROM INSERTED, Emp
    WHERE INSERTED.DNo = Emp.DeptNo AND Emp.Sal > 50K )

• Rules are more complex for the other events
Replication

- Several copies of the same information, typically in a distributed DB
- Synchronized copies: very costly, often unnecessary ⇒ asynchronous techniques to propagate changes
- **Asymmetric replication**
  - one primary copy (where changes are performed) and several secondary copies (read only, updated asynchronously)
  - capture module: monitors changes made by applications to primary copy
  - application module: propagates changes to secondary sites
- **Symmetric replication**
  - copies accept updates asynchronously
  - each have a capture and an application module
  - simultaneous updates may cause loss of consistency

Capture Rules: Example

CREATE RULE Capture1 ON Primary
WHEN INSERTED
THEN INSERT INTO PosDelta
  (SELECT * FROM INSERTED)

CREATE RULE Capture2 ON Primary
WHEN DELETED
THEN INSERT INTO NegDelta
  (SELECT * FROM DELETED)

CREATE RULE Capture3 ON Primary
WHEN UPDATED
THEN INSERT INTO PosDelta (SELECT * FROM INSERTED);
  INSERT INTO NegDelta (SELECT * FROM DELETED);

- Active rules construct the positive and negative deltas, when source data changes; deltas are then applied to secondary copies
Business Rules: Advantages

- Active rules can impose a central consistent behavior independent of transactions that cause their execution.
- Active rules enforce data management policies that no transaction can violate.
- Activities redundantly coded in several application programs with passive DBMSs can be abstracted in a single version.
- **Knowledge independence**: data management policies can evolve by just modifying rules instead of application programs.

Business Rules: Difficulties

- Rule organization and content are often hard to control and to specify declaratively.
- Understanding active rules can be difficult:
  - ♦ rules can react to intricate event sequences
  - ♦ the outcome of rule processing can depend on the order of event occurrences and of rule scheduling
- There are no easy-to-use techniques for designing, debugging, verifying, monitoring sets of rules.
Case Study: Energy Management System

- Example of application modeled with active rules covering the business process: management of the Italian electrical power distribution network.
- Operational network is a forest of trees connecting power distributors to users.
- The operating conditions are monitored constantly with frequent reconfigurations (dynamic structure of the network).
- The topology is modified less frequently (static structure).
- Objective: transfer the exact power from distributors to users through nodes and directed branches connecting pairs of nodes.
- Active rules are used to respond to input transactions asking for
  - reconfigurations due to new users
  - changes in their required power
  - changes in the assignment of wires

EMS Case Study: Database Schema

User(UserId, BranchIn, Power)
  foreign key (BranchIn) references Branch
Branch(BranchId, FromNode, ToNode, Power)
Node(NodeId, BranchIn, Power)
  foreign key (BranchIn) references Branch
Distributor(NodeId, Power, MaxPower)
Wire(WireId, BranchId, WireType, Power)
  foreign key (Branch) references Branch
  foreign key (WireType) references WireType
WireType(WireTypeId, MaxPower)
The network is composed of sites and connections between pairs of sites

- sites comprise
  - power stations (Distributors where power is generated and fed into the network)
  - intermediate nodes (Nodes) where power is transferred to be redistributed across the network
  - final Users of electrical power
- connections are called “branches”
  - class Branch describes all connections between pairs of sites
  - several Wires are placed along the branches
  - wires are of a given WireType, each type carrying a maximum power
  - branches can be dynamically added or dropped to the network

EMS Case Study: Business Rules

- Several user requests are gathered in a transaction
- If the power requested on wires exceeds the maximum power of the wire type, rules change or add wires in the relevant branches
- Rules propagate changes up in the tree adapting the network to new user needs
- A transaction fails if the maximum power requested from some distributor exceeds the maximum power available at the distributor (redesign of the static network is then needed)
- To avoid unnecessary rollbacks, rules propagate reductions of power first, then increases of power
  - This requires setting the order in which the triggers execute
  - Impossible to specify in SQL Server
Connect a New User

- Connecting a new user to node
  ```sql
  create procedure insertUser (@Node char(3), @Power int) as
  declare @User char(3), @Branch char(3), @Wire char(3)
  exec @User = nextUserId
  exec @Branch = nextBranchId
  exec @Wire = nextWireId
  insert into Branch (BranchId, FromNode, ToNode, Power)
  values (@Branch, @User, @Node, @Power)
  insert into Wire (WireId, Branch, WireType, Power)
  values (@Wire, @Branch, 'WT1', @Power)
  insert into User (UserId, BranchIn, Power)
  values (@User, @Branch, @Power)
  ```

- Node to which a user is connected determined by external application: typically its closest node
- WT1 is the “basic” wire type
- nextUserId, nextBranchId, nextWireId: procedures to obtain the next identifier of a user, branch, or wire

Propagation of Power Reduction from a User

- If a user requires less power, propagate the change to its input branch
  ```sql
  create trigger T1_User_Branch on User
  after update as
  if exists ( select * from Inserted I join Deleted D
  on I.UserId = D.UserId where D.Power > I.Power )
  begin
  update Branch
  set Power = Power - ( select D.Power-I.Power
  from Inserted I join Deleted D on I.UserId = D.UserId
  where I.BranchIn = BranchId and D.Power > I.Power )
  where BranchId in ( select BranchIn from Inserted )
  end;
  ```
Propagation of Power Reduction from a Node

- If a node requires less power, propagate the change to its input branch

```sql
create trigger T2_Node_Branch on Node
after update as
if exists ( select * from Inserted I join Deleted D
on I.NodeId = D.NodeId where D.Power > I.Power )
begin
    update Branch
    set Power = Power - ( select D.Power-I.Power
    from Inserted I join Deleted D on I.NodeId = D.NodeId
    where I.BranchIn = BranchId and D.Power > I.Power )
    where BranchId in ( select BranchIn from Inserted )
end;
```

Propagation of Power Reduction from a Branch to a Node

- If a branch connected to a node requires less power, propagate the change to its input node

```sql
create trigger T3_Branch_Node on Branch
after update as
if exists ( select * from Inserted I join Deleted D
on I.BranchId = D.BranchId where D.Power > I.Power
and I.ToNode in ( select NodeId from Node ) )
begin
    update Node
    set Power = Power - ( select D.Power-I.Power
    from Inserted I join Deleted D on I.BranchId = D.BranchId
    where I.ToNode = NodeId and D.Power > I.Power )
    where NodeId in ( select ToNode from Inserted )
end;
```
Propagation of Power Reduction from a Branch to a Distributor

- If a branch connected to a distributor requires less power, propagate the change to the distributor

```sql
create trigger T4_Branch_Distributor on Branch
after update as
if exists ( select * from Inserted I join Deleted D
  on I.BranchId = D.BranchId where D.Power > I.Power
  and I.ToNode in ( select NodeId from Distributor ) )
begin
  update Distributor
  set Power = Power - ( select D.Power-I.Power
    from Inserted I join Deleted D on I.BranchId = D.BranchId
    where I.ToNode = NodeId and D.Power > I.Power )
  where NodeId in ( select ToNode from Inserted )
end;
```

Propagation of Power Increase from a User

- If a user requires more power, propagate the change to its input branch

```sql
create trigger T5_User_Branch on User
after update as
if exists ( select * from Inserted I join Deleted D
  on I.UserId = D.UserId where D.Power < I.Power )
begin
  update Branch
  set Power = Power - ( select D.Power-I.Power
    from Inserted I join Deleted D on I.BranchId = D.BranchId
    where I.BranchIn = BranchId and D.Power < I.Power )
  where BranchId in ( select BranchIn from Inserted )
end;
```
Propagation of Power Increase from a Node

- If a node requires more power, propagate the change to its input branch

```sql
create trigger T6_Node_Branch on Node
after update as
if exists ( select * from Inserted I join Deleted D
    on I.NodeId = D.NodeId where D.Power < I.Power )
begin
    update Branch
    set Power = Power - ( select D.Power-I.Power
        from Inserted I join Deleted D on I.NodeId = D.NodeId
        where I.BranchIn = BranchId and D.Power < I.Power )
    where BranchId in ( select BranchIn from Inserted )
end;
```

Propagation of Power Increase from a Branch to a Node

- If a branch connected to a node requires more power, propagate the change to its input node

```sql
create trigger T7_Branch_Node on Branch
after update as
if exists ( select * from Inserted I join Deleted D
    on I.BranchId = D.BranchId where D.Power < I.Power
    and I.ToNode in ( select NodeId from Node ) )
begin
    update Node
    set Power = Power - ( select D.Power-I.Power
        from Inserted I join Deleted D on I.BranchId = D.BranchId
        where I.ToNode = NodeId and D.Power < I.Power )
    where NodeId in ( select ToNode from Inserted )
end;
```
Propagation of Power Increase from a Branch to a Node

- If a branch connected to a distributor requires more power, propagate the change to its input distributor.

```sql
create trigger T8_Branch_Distributor on Branch
after update as
if exists ( select * from Inserted I join Deleted D
  on I.BranchId = D.BranchId where D.Power < I.Power
  and I.ToNode in ( select NodeId from Distributor ) )
begin
  update Distributor
  set Power = Power - ( select D.Power-I.Power
    from Inserted I join Deleted D on I.BranchId = D.BranchId
    where I.ToNode = NodeId and D.Power < I.Power )
  where NodeId in ( select ToNode from Inserted )
end;
```

Excess Power Requested from a Distributor

- If the power requested from a distributor exceeds its maximum, rollback the entire transaction.

```sql
create trigger T9_Distributor on Distributor
after update as
if exists ( select * from Inserted D where D.Power>D.MaxPower )
begin
  raiserror 13000 'Maximum capacity of the distributor exceeded'
  rollback
end;
```
Propagate Power Change from a Branch to Its Wires

- If the power of a branch is changed, distribute the change equally on its wires

create trigger T10_Branch_Wire on Branch
after update as
begin
update Wire
set Power = Power - ((select D.Power - I.Power
from Inserted I join Deleted D on I.BranchId = D.BranchId
where I.BranchId = Branch) /
(select count(*) from Wire W join Inserted I
on I.BranchId = W.Branch where W.Branch = Branch))
where Branch in (select BranchId from Inserted)
end;

Change Wire Type if Power Passes Threshold

- If the power on a wire goes above the allowed threshold, change the wire type

create trigger T11_Wire_Wire on Wire
after insert, update as
if exists (select * from Inserted W join WireType WT
on WireType = WireTypeId where W.Power > WT.MaxPower and exists (select * from WireType WT1 where WT1.MaxPower > W.Power))
begin
update Wire set WireType = (select WireTypeId from WireType WT
where WT.MaxPower >= Power and not exists (-- it is supposed that no two WireTypes have the same MaxPower
select * from WireType WT1 where WT1.MaxPower < WT.MaxPower
and WT1.MaxPower >= Power))
where WireId in (select WireId from Inserted W join WireType WT
on WireType = WireTypeId where W.Power > WT.MaxPower
and exists (select * from WireType WT1
where WT1.MaxPower > W.Power))
end;
Add a Wire to a Branch

- If there is no suitable wire type, add another wire to the branch

```sql
create trigger T12_Wire_Wire on Wire
after insert, update as
if exists ( select * from Inserted W join WireType WT
  on WireType = WireTypeId where W.Power > WT.MaxPower
  and W.Power > ( select max(MaxPower) from WireType ) )
begin
  declare @nextWire char(3), @BranchId char(3), @WireId char(3),
  @Power real, @MaxPower real
  declare wires_cursor cursor for
  select W.WireId, W.BranchId, W.Power, WT.MaxPower
  from Inserted W join WireType WT on WireType = WireTypeId
  where W.Power > WT.MaxPower
  and W.Power > ( select max(MaxPower) from WireType )
  open wires_cursor
  fetch next from wires_cursor
  into @WireId, @BranchId, @Power, @MaxPower
  while @@FETCH_STATUS = 0
  begin
    exec @nextWire = nextWireId
    insert into Wire (WireId, BranchId, WireType, Power)
    values( @nextWire, @BranchId, 'WT1', @Power-0.8*@MaxPower )
    fetch next from wires_cursor
    into @WireId, @BranchId, @Power, @MaxPower
  end
  close wires_cursor
  deallocate wires_cursor
  update Wire set Power = ( select 0.8*MaxPower from WireType WT
  where WT.WireTypeId = WireType )
  where Power > ( select max(MaxPower) from WireType )
end;
```

Add a Wire to a Branch (cont.)

```sql
open wires_cursor
fetch next from wires_cursor
  into @WireId, @BranchId, @Power, @MaxPower
while @@FETCH_STATUS = 0
begin
  exec @nextWire = nextWireId
  insert into Wire (WireId, BranchId, WireType, Power)
  values( @nextWire, @BranchId, 'WT1', @Power-0.8*@MaxPower )
  fetch next from wires_cursor
  into @WireId, @BranchId, @Power, @MaxPower
end
close wires_cursor
deallocate wires_cursor
update Wire set Power = ( select 0.8*MaxPower from WireType WT
  where WT.WireTypeId = WireType )
  where Power > ( select max(MaxPower) from WireType )
end;
```
Active Databases: Summary

- Active technology has been growing
  - availability of research prototypes and commercial products
  - wide range of applications
  - much research activity
- But, relatively little impact in practice
  - DB administrator and DB developers make little use of triggers
  - general fear: use of rules entails
    - lower performance
    - loss of control
- Consensus: design methods and tools are needed for help in designing, debugging, and monitoring sets of active rules