Databases 2

6

Active Databases

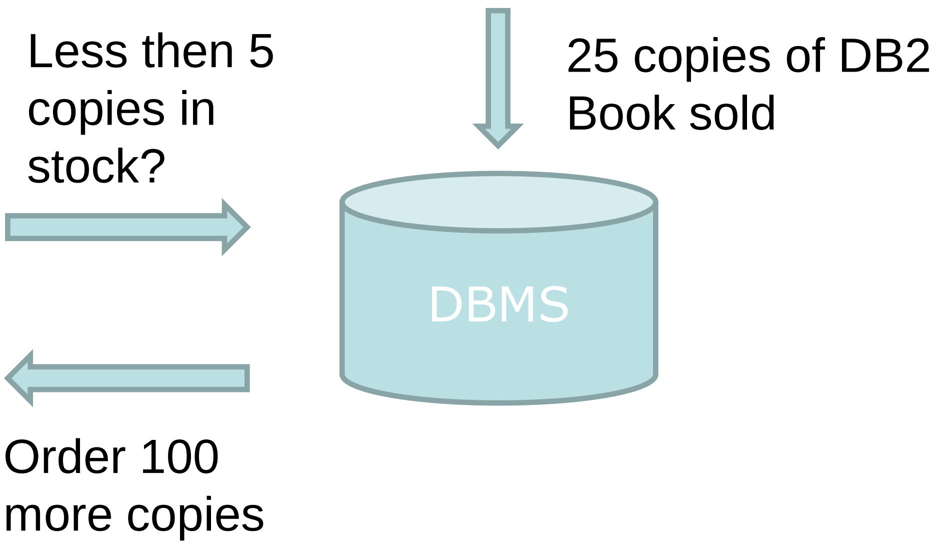
Active Databases

- Databases that support active rules
  (also called *triggers*)

- Outline:
  - Trigger definition in SQL:1999
  - Properties of trigger-based systems
    - Termination, confluence, design methods, …
    - Evolution of triggers
  - Several examples
  - Trigger definition in DB2 and Oracle

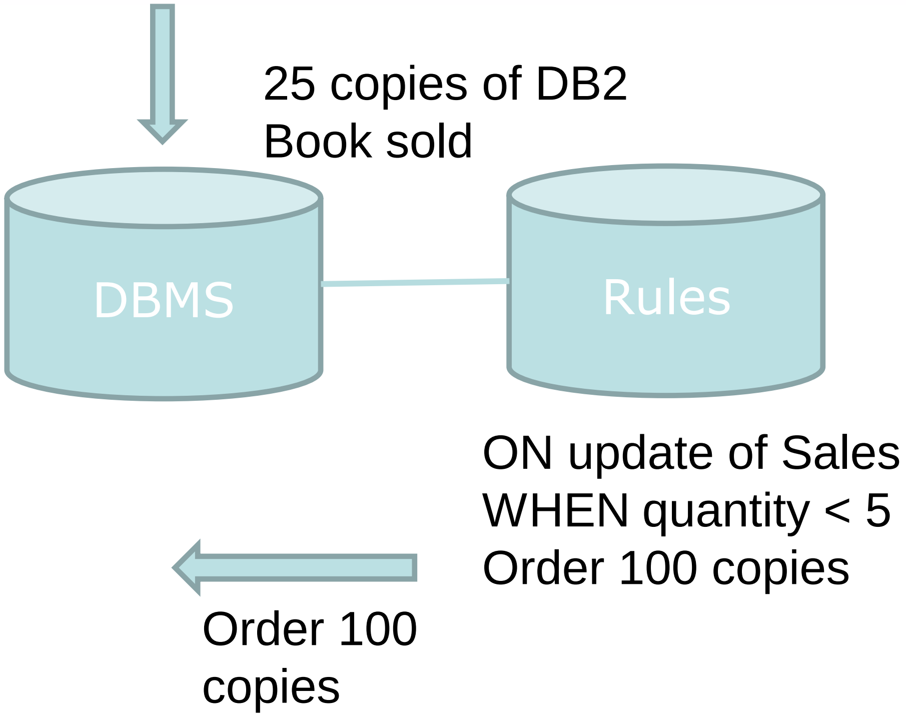
Passive database

- Periodical polling
  - Too frequent: expensive
  - Infrequent: miss the right time to reach
- The polling must be done for all items in stock



Active database

- Triggers define actions when situations occur
- Actions are usually database updates



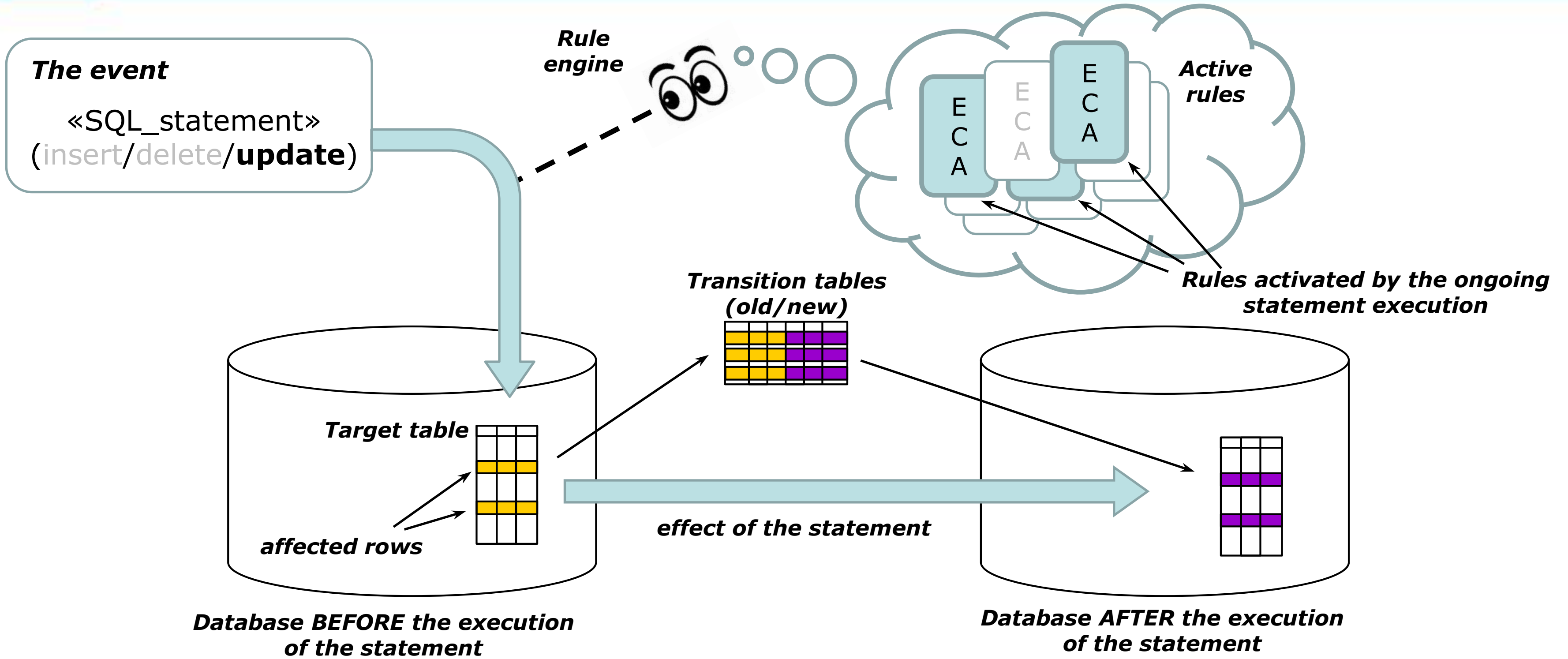
The Trigger Concept

- **ECA** Paradigm: **E**vent-**C**ondition-**A**ction
  - **whenever** an event ***e*** occurs
  - **if** a condition ***c*** is true
  - **then** an action ***a*** is executed
- An effective means to implement reactive computations
- Other examples of reactive behaviors in the DBMS world:
  - Integrity constraints (and reaction policies)
  - Datalog rules
  - Business rules within database schemas
- Problem: it is difficult to implement complex, sophisticated database applications using triggers

Event-Condition-Action

- **Event**
  - Normally a modification of the database status: **`insert`**, **`delete`**, **`update`**
  - When the event occurs, the trigger is ***activated***
- **Condition**
  - A predicate that identifies those situations in which the execution of the trigger's action is required
  - When the condition is evaluated, the trigger is ***considered***
- **Action**
  - A generic update statement or a stored procedure
  - When the action is elaborated, the trigger is ***executed***
- DBMSs already provide all the required components. Support for triggers just requires their integration

The big picture



Triggers in SQL:1999, Syntax

- SQL:1999 (aka SQL-3) was strongly influenced by DB2 (IBM)
  - the other systems, initially not fully compliant (as they exist since the mid eighties), tend to the standard
- Each trigger is characterized by (at least):
  - a name
  - an execution mode (**`before`** or **`after`**)
  - a monitored event (typically: **`insert`**, **`delete`**, or **`update`**)
  - the name of the target (monitored) table
  - a granularity (**`statement`**-level or **`row`**-level)
  - names and aliases for transition values and transition tables
  - an action
  - creation timestamp

Triggers in SQL:1999, Syntax

```
create trigger <TriggerName>
{ before | after }
{ insert | delete | update [of <Column>] } on <Table>
[ referencing { [ old table [as] <OldTableAlias> ]
                [ new table [as] <NewTableAlias> ] |
                [ old [row] [as] <OldTupleName> ]
                [ new [row] [as] <NewTupleName> ] } ]
[ for each { row | statement } ]
[ when <Condition> ]
<SQLProceduralStatement>
```

Execution modes: `before` or `after`

- **BEFORE**
  - The trigger is considered (and possibly executed) **before the event** is applied (i.e., <u>before the database status change</u>)
  - *Safeness constraint*: before triggers cannot update the database
    - at most, they can affect ("condition") the transition variables in row-level granularity (*set new.t=<expr>*)
  - Typically, this mode is used to check and validate a modification before it takes place, and possibly condition the modification itself
- **AFTER**
  - The trigger is considered (and possibly executed) **after the event**
    - It is the most common mode, suitable for most applications

Granularity of events

- **Statement-level** granularity (default: **`for each statement`**)
  - The trigger is considered (and possibly executed) only once for each activating **statement**, independently of the number of affected tuples in the target table (even if no tuple is affected!)
  - Closer to the traditional approach of SQL statements, which are normally set-oriented
- **Row-level** granularity (keyword: **`for each row`**)
  - The trigger is considered (and possibly executed) once for each **tuple** affected by the activating statement
  - Writing row-level triggers is simpler, but can be less efficient

The `referencing` clause

- Its syntax descends from the chosen granularity
  - If it is row-level, two ***transition variables*** (**`old`** and **`new`**) represent the value respectively prior to and following the modification of the row (i.e., tuple) under consideration
  - If it is statement-level, two ***transition tables*** (**`old table`** and **`new table`**) contain respectively the old and the new value of all the affected rows (tuples)
- Variables **`old`** and **`old table`** are undefined in triggers whose event is **`insert`**
- Variables **`new`** and **`new table`** are undefined in triggers whose event is **`delete`**
- Transition variables and transition tables enable tracking of the changes that activate triggers, and are crucial to efficiency

Example of a row-level trigger

```
create trigger AccountMonitor
after update of Balance on Account
for each row
when new.Balance > old.Balance
insert into IncomingPayments
   values( new.AccNumber, new.Balance-old.Balance,
           sysdate() )
```

Example of a statement-level trigger

```
create trigger FilingOfDeletedInvoices
after delete on Invoice
referencing old table as OldInvoiceSet
insert into DeletedInvoices
   ( select *
     from OldInvoiceSet )
```

Example of a trigger in before mode

- Prevents salaries to be increased by more than 20% within a single update operation
- Presented in two versions: 1. before mode 2. after mode
- 1. "**conditioner**" (acts before the update and integrity checking)

```
create trigger Max20percent_before
before update of Salary on Employee
for each row
when new.Salary > old.Salary * 1.2
set new.Salary = old.Salary * 1.2
```

the transition variable is modified

The same effect in after mode

- 2. "**re-installer**" (acts after the update)

```
create trigger Max20percent_after
after update of Salary on Employee
for each row
when new.Salary > old.Salary * 1.2
update Employee
  set Salary = old.Salary * 1.2
    where RegNum = new.RegNum
```

the database is modified

Execution of Multiple Triggers: Conflicts

- If several triggers are associated with the same event, SQL:1999 prescribes the following execution policy
  - BEFORE triggers (**statement**-level first, and then **row**-level) are considered and possibly executed
  - The modification is applied and the integrity constraints defined on the DB are checked
  - AFTER triggers (**row**-level first, and then **statement** level) are considered and possibly executed
- If there are several triggers in the same category, the order of execution depends on the system implementation (e.g., based on the definition time (older triggers have higher priority))

Recursive Execution Model in SQL:1999

- Triggers are handled within Trigger Execution Contexts (TECs)
- The execution of a trigger action may activate other triggers, that are to be evaluated within new, “inner” TECs
  - The “outer” TEC is saved, the inner one is built, and its trigger is executed
    - this is a **recursive** process: for a transaction, at any time, there may be several nested TECs, stored in a stack, only the topmost being active
  - At the end of each inner TEC’s execution, the outer one is restored and its execution is resumed
- The TEC of row-level triggers accounts for rows that were already considered and rows still to be processed
- Any failure during a chain of activations due to a statement S causes the rollback of S and of all the changes performed by the chain
  - Safeness heuristics: the execution typically halts when a given recursion depth is reached, rising a “nontermination exception”

Example: Salary Management

Employee

RegNum	Name	Salary	DeptN	ProjN
50	Smith	5.900	1	20
51	Black	5.600	1	10
52	Jones	5.000	1	20

Department

DeptNum	MGRRegNum
1	50

Project

ProjNum	Crucial
10	no
20	no

Example Trigger T1: Bonus

Event: `update` of the `Crucial` attribute in table `Project`

Condition: New value: `Crucial = 'yes'`

Action: Increase by 10% the salary of the employees involved in the project that becomes crucial

```
create trigger Bonus_T1
after update of Crucial on Project
for each row
when new.Crucial = 'yes' and old.Crucial = 'no'
update Employee
   set Salary = Salary * 1.10
   where ProjNum = new.ProjNum;
```

Example Trigger T2: CheckIncrement

Event: `update` of `Salary` in `Employee`
Condition: The new salary is greater than the manager’s salary
Action: Decrease salary and make it the same as the manager’s

```
create trigger CheckIncrement_T2
after update of Salary on Employee
for each row
declare X number;
begin
   select Salary into X
   from Employee join Department D on RegNum = D.MGRRegNum
   where D.DeptNum = new.DeptN;

   if( new.Salary > X ) update Employee set Salary = X
                        where RegNum = new.RegNum;       endif;
end;
```

Example Trigger T3: CheckDecrement

Event: `update` of `Salary` in `Employee`
Condition: Decrement greater than 3%
Action: Decrement salary only by 3%

```
create trigger CheckDecrement_T3
after update of Salary on Employee
for each row
when new.Salary < old.Salary * 0.97
update Employee
   set Salary = old.Salary * 0.97
   where RegNum = new.RegNum;
```

Activation of T1

```
update Project
  set Crucial = 'yes'
    where ProjNum = 10
```

Project

ProjNum	Crucial
10	**yes**
20	no

Event: update of `Crucial` in `Project`

Condition: true

Action: increase Black's salary by 10%

Employee

RegNum	Name	Salary	DeptN	ProjN
50	Smith	5.900	1	20
51	Black	**6.160**	1	10
52	Jones	5.000	1	20

Activation of T2

Event: update of **`Salary`** in **`Employee`**

Condition: true (Black’s salary is greater than Smith’s)

Action: Black’s salary is set to Smith’s

RegNum	Name	Salary	DeptN	ProjN
50	Smith	5.900	1	20
51	Black	**5.900**	1	10
52	Jones	5.000	1	20

- T2 is activated again – the condition in false (not increased)
- T3 is activated

Activation of T3

Event: update of **`Salary`** in **`Employee`**

Condition: true (Black’s salary was decreased by more than 3%)

Action: Black’s salary is decreased by only 3%

RegNum	Name	Salary	DeptN	ProjN
50	Smith	5.900	1	20
51	Black	**5.975,20**	1	10
52	Jones	5.000	1	20

- T3 is activated again – the condition is false (not decreased)
- T2 is activated again – the condition is true (increased)

Activation of T2

RegNum	Name	Salary	DeptN	ProjN
50	Smith	5.900	1	20
51	Black	**5.900**	1	10
52	Jones	5.000	1	20

Activation of T3

- The trigger condition is false
  - This time, the salary was decreased by less than 3% - T3 is overall ineffective!
- Trigger activation has reached termination

Design - Trigger Properties

- It is important to ensure that **interferences** among triggers and **chain activations** do not produce undesired system behaviors
- Three classical properties
  - **Termination**: for any initial state and any sequence of modifications, a final state is always produced (infinite activation cycles are not possible)
  - Confluence: triggers terminate and produce a unique final state, independent of the order in which triggers are executed
    - Meaningful only if there is nondeterminism in the activation
  - Determinism of observable behavior: triggers are confluent and produce the same sequence of messages
- Termination is by far the most important property

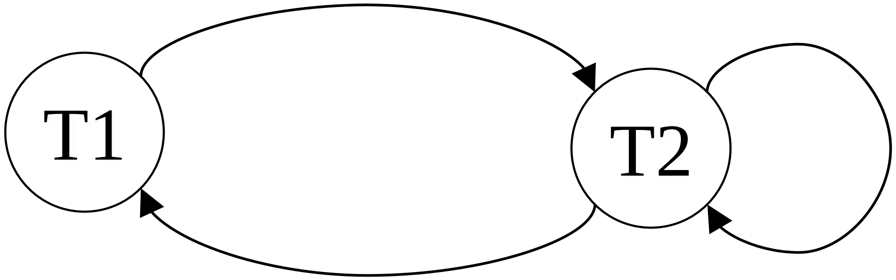
Termination Analysis

- There exist several tools, most of which based on graphs
- The simplest abstraction is the **triggering graph**
  - A node i for each trigger t_i
  - An arc from a node i to a node j if the execution of trigger t_i's action **may** activate trigger t_j
  - The graph is built with a simple syntactic analysis
- If the graph is acyclic, the system is guaranteed to terminate
  - There cannot be infinite trigger sequences
- If the graph has some cycles, it *may* be non-terminating
  - but it might as well be terminating (cfr.: recursion in PLs)

Example with Two Triggers

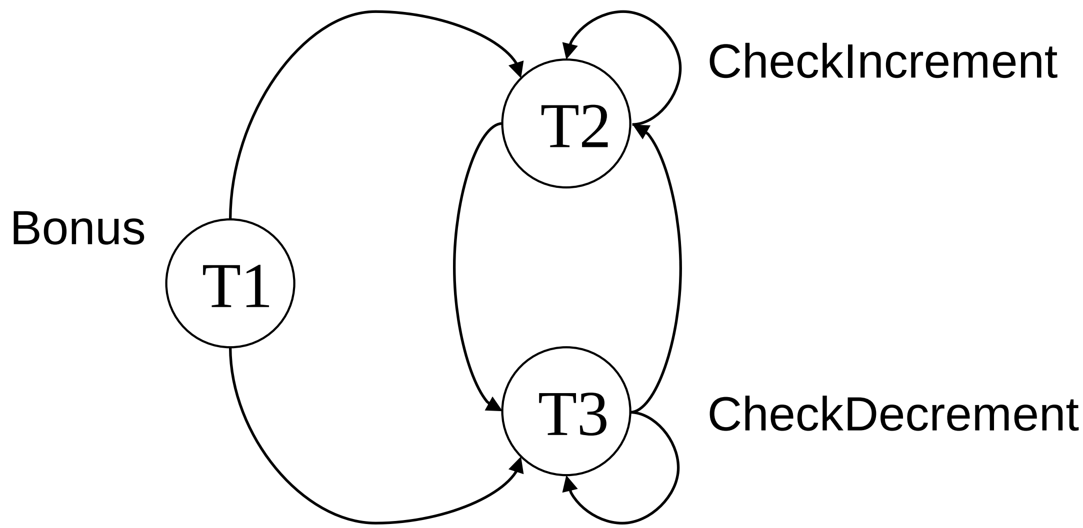
```
T1:     create trigger AdjustContributions
        after update of Salary on Employee
        referencing new table as NewEmp
        update Employee
        set Contribution = Salary * 0.8
        where RegNum in ( select RegNum from NewEmp )

T2:     create trigger CheckOverallBudgetThreshold
        after update on Employee
        when 50000 < ( select sum(Salary+Contribution)
                       from Employee )
        update Employee
        set Salary = 0.9 * Salary
```

Triggering Graph for the previous triggers



- There are two cycles, but the system is terminating
- Can be changed into non-terminating, e.g., by inverting the comparison in T2’s condition

Termination Graph for Salary Management



- The graph is cyclic, but the repeated execution of the triggers reaches termination anyway

A nonconfluent trigger

```
create trigger Bonus_Alternative
after update of Crucial on Project
for each row
declare X number
when new.Crucial = 'yes'
begin select sum(Residual) into X from Budget
      if( X > 30.000 )
      begin update Residual
             set Residual = Residual - ( select sum(salary * 0.1)
                                          from Employee
                                          where ProjN = new.ProjN );
              where ProjN = new.ProjN
            update Employee
             set Salary = Salary * 1.1
              where ProjN = new.ProjN;
      end;
end;
```

Techniques and methodologies for trigger design

- Proposed for smaller- and larger-scale design
  - Small-scale design: the best option is to give a try and then rely on existing analysis tools
  - Large-scale: there are specific methodologies
- Modularization:
  - Triggers are clustered in modules, each with a specific purpose
    - Facilitates the proof that interferences are harmless and that each module reaches its objective
    - If so, the system is overall correct

Problems in Designing Trigger Applications

- Triggers add powerful data management capabilities in a transparent and reusable manner
  - Databases can be enriched with “business and management rules” that would otherwise be distributed over all applications
- However, understanding the interactions between triggers is rather complex: triggers are an underexploited feature
- DBMS vendors use triggers to implement internal services, by introducing mechanisms for their automatic generation
  - Examples:
    - Constraint management
    - Data replication
    - View maintenance

Applications of Active Databases

- **Internal** rules (system-generated and not visible to users)
  - Integrity constraint management
  - Computation of derived and replicated data
  - Versioning management, privacy, security
  - Action logging, event recording
- **External** rules (generated by database administrators, express application-specific knowledge)
  - Personalization, adaptation
  - Context-awareness
  - Business rules

Referential Integrity Management

- *Repair strategies* for violations of referential integrity constraints
  - The constraint is expressed as a predicate in the condition part

Ex:
```
CREATE TABLE Employee (
… …
FOREIGNKEY(DeptN) REFERENCES Department(DeptNum)
  ON DELETE SET NULL  ON UPDATE CASCADE,
… … );
```

- Operations that can violate this constraint:
  - **INSERT** into Employee
  - **UPDATE** of Employee.DeptN
  - **UPDATE** of Department.DeptNum
  - **DELETE** from Department

Actions in the Employee Table

Event: insert into **Employee**
Condition: the new **DeptN** value is not in the **Dept** table
Action: **no action** policy: insertion is inhibited, reporting error

```
create trigger CheckEmpDept
before insert on Employee
for each row
when not exists ( select * from Department
                  where DeptNum = new.DeptN )
raise_application_error(-20000, ‘Invalid Department‘);
```

- The trigger for the *update* of **DeptN** in **Employee** is analogous (identical but for **before update** instead of **before insert**)

  [*à-la-Oracle: rollback is automatically done by* `raise_application_error`]

Deletion in the Department Table

Event: delete from **`Department`**
Condition: the deleted **`DeptNum`** is used in the **`Employee`** table
Action: **set null** policy (the employee's **`DeptN`** is set to null)

```
create trigger CheckDeptDeletion
after delete on Department
for each row
when exists ( select * from Employee
              where DeptN = old.DeptNum )
update Employee set DeptN = null
  where DeptN = old.DeptNum;
```

(note: condition could be omitted)

Updates in the Department Table

Event: update of `DeptNum` in `Department`
Condition: the old `DeptNum` value is used in the `Employee` table
Action: **cascade** policy (`DeptN` in Employee is also modified)

```
create trigger CheckDeptUpdate
after update of DeptNum on Department
for each row
when exists ( select * from Employee
              where DeptN = old.DeptNum)
update Employee set DeptN = new.DeptNum
   where DeptN = old.DeptNum;
```

(note: condition could be omitted)

Triggers for Replicas and View Maintenance

- Consistency of views w.r.t. the tables on which they are defined
  - Base table updates must be propagated to views
- Materialized view maintenance is typically managed via triggers
- Also: replication management:

```
CREATE MATERIALIZED VIEW EmployeeReplica
REFRESH FAST AS
SELECT * FROM
DBMaster.Employee@mastersite.world;
```

Triggers for Materialized View Maintenance

- Materialized views are helpful whenever a query is performed more frequently w.r.t. the updates that may change its result
- A naïve approach consists in re-computing the view whenever the base table is modified
- Example: overall personnel cost of each department:

```
CREATE MATERIALIZED VIEW PersonnelDepCost(DpN,SalarySum) AS
select DeptN, sum(Salary)
from Employee
group by DeptN;
```

Triggers for Materialized View Maintenance

- Triggers implementing the naïve approach (**statement-level**)

```
create trigger NaiveRecompute_upd(|_del|_ins)
after update(|delete|insert) on Employee
begin
  delete from PersonnelDepCost;
  insert into PersonnelDepCost
    select DeptN, sum(Salary)
    from Employee
    group by DeptN;
end;
```

Three almost identical triggers: whatever the modification is, the whole materialization is dropped and reinstalled

Triggers for Materialized View Maintenance

- However, not all modifications affect the view, and most modifications only affect a small part of the materialized data:
  - Update **`RegNum`**, **`Name`**, or **`ProjN`**: no effect
  - Insertion of a new Employee: only affects one tuple
  - Deletion of an Employee: only affects one tuple
  - Update of one **`Salary`**: only affects one tuple in **`PersonnelDepCost`**
  - Update of one **`DeptN`**: only affects two tuples in **`PersonnelDepCost`**
- In all these cases, and whenever the computational effort required to deal with the delta in the view is significantly smaller than that of fully recomputing the view, an ***incremental*** approach is preferrable
  - A higher number of (more specific) triggers may be required

Materialized View Maintenance: Incremental approach

- Incremental approach: Insertion and deletion of employees

```
create trigger Incremental_InsEmp
after insert on Employee
for each row
update PersonnelDepCost
 set SalarySum = SalarySum + new.Salary
  where DpN = new.DeptN;
```

```
create trigger Incremental_DelEmp
after delete on Employee
for each row
update PersonnelDepCost
 set SalarySum = SalarySum - old.Salary
  where DpN = old.DeptN;
```

***row-level** triggers !*

If more than one employee is inserted(deleted) by the same SQL command, each affected tuple affects only one department in the materialized view

Materialized View Maintenance: Incremental approach

- Incremental approach: update of **Salary**

```
create trigger Incremental_SalaryUpdate
after update of Salary on Employee
for each row
update PersonnelDepCost
 set SalarySum = SalarySum + new.Salary - old.Salary
  where DpN = new.DeptN;
```

Applies the “salary delta”...

...only to the department to which the updated employee is affiliated (using old.DeptN would be equally correct, as this is not changed)

Materialized View Maintenance: Incremental approach

- Incremental approach: update of **DeptN** (moving the employee)

```
create trigger Incremental_DeptChange
after update of DeptN on Employee
for each row
begin
  update PersonnelDepCost
    set SalarySum = SalarySum + new.Salary
      where DpN = new.DeptN;
  update PersonnelDepCost
    set SalarySum = SalarySum - old.Salary
      where DpN = old.DeptN;
end;
```

The sum is incremented (resp. decremented) in the new (resp. old) department

View Maintenance: the role of Integrity

- The previous incremental triggers are correct only if
  - The modifications do not violate the integrity constraints
  - The previous materialization is already correct
  - The other tables (other than Employee) are not modified

  otherwise, the resulting materialization may not be aligned with the base table
- Instead, the "naïve" approach is less error prone w.r.t. this aspect
- Example: we didn't consider the creation of a new department!
  - Employees moved to the new department would subtract their salary from their old affiliation but wouldn't add it to the new one, not mentioned in the materialization
  - A dedicated trigger should create a new tuple in the materialization

View Maintenance: the role of Integrity

- A dedicated trigger should create a new tuple in the materialization:

```
create trigger Incremental_NewDept
after insert on Department
insert into PersonnelDepCost
 select DeptNum , 0
 from new table;
```

This statement -level trigger is guaranteed to activate before the row-level ones: it is correct to insert the new tuple with 0 as initial value for SalarySum

- Also, we are assuming that there exists a referential integrity from Employee(DeptN) to Department(DeptN)
  - Without it, it would be possible to move an employee to a “nonexistent” department, and the contribution of the moved salary would be “lost” by the incremental triggers implemented so far
    - but still not by the “naïve” ones, that would “blindly” rebuild the groups

Recursion Management

- Triggers for recursion management
  - Recursion not yet supported by most DBMSs
- Ex.: representation of a hierarchy of products
  - Each product is characterized by a **super-product** and by a depth **level** in the hierarchy
  - Can be represented by a recursive view (**`with recursive`** construct in SQL:1999)
  - Alternatively: use triggers to build and maintain the hierarchy

`Product(Code,Name,Description,SuperProduct,Level)`

- Hierarchy represented by **`SuperProduct`** and **`Level`**
- Products not contained in other products have:

  `SuperProduct = NULL` and **`Level = 0`**

Deletion of a Product

- In case of product deletion, all its sub-products must be deleted as well

```
create trigger DeleteProduct
after delete on Product
for each row
delete from Product
  where SuperProduct = old.Code;
```

Insertion of a New Product

- In case of insertions, the appropriate **`Level`** must be calculated

```
create trigger ProductLevel
after insert on Product
for each row
if( new.SuperProduct is not null )
  update Product
     set Level = 1 + ( select Level from Product
                       where Code = new.SuperProduct )
        where Code = new.Code;
else
  update Product
     set Level = 0
        where Code = new.Code;
endif;
```

Insertion of a New Product

- In **before** mode it is even simpler:

```
create trigger ProductLevel
before insert on Product
for each row
if( new.SuperProduct is not null )
   set new.Level = 1 + ( select Level from Product
                         where Code = new.SuperProduct )
else
    set new.Level = 0
endif;
```

Access Control

- Triggers can be used to strengthen access control
- It is convenient to define only those triggers that correspond to conditions that can't be directly verified by the DBMS
- Using BEFORE triggers with STATEMENT granularity gives the following advantages
  - Access control is performed before the triggering event is executed
  - Access control is executed only once and not for each tuple affected by the trigger event

ForbidSalaryUpdate Trigger

```
create trigger ForbidSalaryUpdate
before insert on Employee
DECLARE not_weekend EXCEPTION; not_workingHours EXCEPTION;
begin
  if( to_char(sysdate, 'dy') = 'sat'        /* if weekend*/
      OR to_char(sysdate, 'dy') = 'sun' )
    raise not_weekend;
  endif;
  if( to_char(sysdate,'HH24') < 8         /* if not in working */
      OR to_char(sysdate,'HH24') > 18 ) /*   hours (8-18)     */
    raise not_workingHours;
  endif;
end;
```

ForbidSalaryUpdate Trigger (cont'd)

```
exception
  when not_weekend
    then raise_application_error(-20324,"cannot
      modify Employee table during week-end");
  when not_workingHours
    then raise_application_error(-20325,"cannot
      modify Employee table outside working hours");
end;
```

Evolution of active databases

- Execution modes (immediate, deferred, detached)
- New events (system-defined, temporal, user-defined)
- Complex events and event calculus
- Instead-of clause
- Rule administration: priorities, grouping, dynamic activation and deactivation
- Variations introduced by vendors, an example: Oracle

Execution Modes

- The execution mode describes the connection between the activation (event) and the consideration and execution phases (condition and action)
- Condition and action are always evaluated together
- Normally the trigger is **Immediate**: considered and executed with the activating event
- Alternative execution modes:
  - **Deferred**: the trigger is handled at the end of the transaction
    - Example: triggers that check satisfaction of integrity constraints that require the execution of several operations
  - **Detached**: the trigger is handled in a separate transaction
    - Example: efficient management of variations of stock indices values after several exchanges

Extended Events/1

- System events and DDL commands
  - System: server-error, shutdown, etc.
  - DDL: authorization updates
  - In both cases some DBMSs already have these services that perform complex monitoring
- Temporal events (also periodical events)
  - Example: on July 23rd 2006 at 12, every day at 4
  - Useful for several applications
  - Difficult to integrate them because they are in an autonomous transactional context
  - They can be simulated via software components outside the DBMS that use time management services from the operating system

Extended Events/2

- “User-defined” events
  - Example: “TemperatureTooHigh”
  - Useful in some applications, but normally not offered
  - They too can be easily simulated
- Queries
  - Example: who reads the salaries
  - Normally too heavy to handle

Event expressions

- Boolean combinations of events
  - SQL:1999 allows the specification of several events for a trigger, in disjunction
    - Any event among these is sufficient
  - Some researchers proposed more complex composition models
    - Very complex to handle
    - No strong motivation for introducing them

Instead of clause

- Alternative to **`before`** and **`after`**
- Another operation than the one that activated the event is executed
- Very dangerous semantics (the application does one thing, the system does another thing)
- Implemented in several systems, often with strong limitations
  - In Oracle it can only be used for updates on views, so as to solve the view update problem when there is ambiguity

Priorities, Activations, and Groups

- Definition of priority
  - Allows specifying the execution order of triggers when there are several triggers activated at the same time
  - SQL:1999 states an order based on the execution mode and granularity; when these coincide, the choice depends on the implementation
- Activation/deactivation of triggers
  - Not in the standard, but often available
- Organization of triggers in groups
  - Some systems offer trigger grouping mechanisms, so as to activate/deactivate by groups

Proprietary limitations and extensions: Oracle

- Oracle follows a different syntax (multiple events allowed, no table variables, when clause only legal with row-level triggers)

  create trigger *TriggerName*
  { **before** | **after** } *<event>* [, *<event>* [, *<event>*]]
  [[**referencing** [**old** [**row**] [**as**] *OldTupleName*]
  [**new** [**row**] [**as**] *NewTupleName*]]
  for each row
  [**when** *SQLPredicate*]]
  PL/SQLStatements

  <event> ::= { **insert** | **delete** | **update** [**of** *Column*] } **on** *Table*

- They have also a rather different conflict semantics, and no limitation on the expressive power of the action of before triggers

Conflicts between Triggers in Oracle

- If several triggers are associated to the same event, ORACLE has the following policy:
  - BEFORE statement-level triggers are executed
  - BEFORE row-level triggers are executed
  - The modification is applied and the integrity constraints defined on the DB are checked
  - AFTER row-level triggers are executed
  - AFTER statement-level triggers are executed
- If there are several triggers belonging to the same category, the order of execution depends on the creation time of the trigger
- "*Mutating table exception*": occurs when the chain of triggers activated by a before trigger T tries to change the state of T's target table. Forces a statement rollback.

Trigger syntax in MySQL

CREATE
[DEFINER = { *user* | CURRENT_USER }]
TRIGGER *trigger_name*
trigger_time *trigger_event* ON *tbl_name*
FOR EACH ROW
[*trigger_order*]
trigger_body

Trigger syntax in PostreSQL

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF }
{ event [ OR ... ] }
    ON table
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY
     IMMEDIATE | INITIALLY DEFERRED } ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:
    INSERT,  UPDATE [ OF column_name [, ... ] ],  DELETE
    TRUNCATE (only for each statement)
```