INFOH415 - Advanced Databases

Jose Antonio Lorencio Abril

Fall 2022

Professor: Esteban Zimanyi

Student e-mail: jose.lorencio.abril@ulb.be

This is a summary of the course *Advanced Databases* taught at the Université Libre de Bruxelles by Professor Esteban Zimanyi in the academic year 22/23. Most of the content of this document is adapted from the course notes by Zimanyi, [1], so I won't be citing it all the time. Other references will be provided when used.

Contents

List of Figures

List of Tables

List of Algorithms

Part I
Active Databases

1 Introduction

Traditionally, DBMS are **passive**, meaning that all actions on data result from explicit invocation in application programs. In contrast, **active DMBS** can perform actions automatically, in response to monitored events, such as updates in the database, certain points in time or defined events which are external to the database.

Integrity constraints are a well-known mechanism that has been used since the early stages of SQL to enhance integrity by imposing constraints to the data. These constraints will only allow modifications to the database that do not violate them. Also, it is common for DBMS to provide mechanisms to store procedures, in the form of precompiled packets that can be invoked by the user. These are usually caled **stored procedure**.

The active database technology make an abstraction of these two features: the **triggers**.

Definition 1.1. A **trigger** or, more generally, an **ECA rule**, consists of an event, a condition and a set of actions:

- **Event**: indicates when the trigger must be called.
- **Condition**: indicates the checks that must be done after the trigger is called. If the condition is fulfilled, then the set of actions is executed. Otherwise, the trigger does not perform any action.
- **Actions**: performed when the condition is fullfilled.

Example 1.1. A conceptual trigger could be like the following:

Event	A customer has not paid 3 invoices at the due date.
Condition	If the credit limit of the customer is less than 20000€.
Action	Cancel all curernt orders of the customer.

There are several aspects of the semantics of an applications that can be expressed through triggers:

- Static constraints: refer to referential integrity, cardinality of relations or value restrictions.
- Control, business rules and workflow management rules: refer to restrictions imposed by the business requirements.
- Historical data rules: define how historial data has to be treated.
- Implementation of generic relationships: with triggers we can define arbitrarily complex relationships.
- Derived data rules: refer to the treatment of materialized attributes, materialized views and replicated data.
- Access control rules: define which users can access which content and with which permissions.
- Monitoring rules: assess performance and resource usage.

The benefits of active technology are:

- Simplification of application programs by embedding part of the functionality into the database using triggers.
- Increased automation by the automatic execution of triggered actions.
- Higher reliability of data because the checks can be more elaborate and the actions to take in each case are precisely defined.
- Increased flexibility with the possibility of increasing code reuse and centralization of the data management.

2 Representative Systems and Prototypes

Even though this basic model is simple and intuitive, each vendor proposes its own way to implement triggers, which were not in the SQL-92 standard. We are going to study Starbust triggers, Oracle triggers and DB2 triggers.

2.1 Starbust

Starbust is a Relational DBMS prototype developed by IBM. In Starbust, the **triggers** are defined with the following definition of their components:

- **Event**: events can refer to data-manipulation operations in SQL, i.e. INSERT, DELETE or UPDATE.
- **Conditions**: are boolean predicates in SQL on the current state of the database after the event has occurred.
- **Actions**: are SQL statements, rule-manipulation statements or the ROLLBACK operation.

Example 2.1. *'The salary of employees is not larger than the salary of the manager of their department.'*

The easiest way to maintain this rule is to rollback any action that violates it. This restriction can be broken (if we focus on modifications on the employees only) when a new employee is inserted, when the department of an employee is modified or when the salary of an employee is updated. Thus, a trigger that solves this could have these actions as events, then it can check whether the condition is fullfilled or not. If it is not, then the action can be rollback to the previous state, in which the condition was fullfilled.

```
CREATE RULE Mgrsals ON Emp
WHEN INSERTED, UPDATED(Dept), UPDATED(Salary)
IF EXISTS (
  SELECT *
  FROM Emp E, Dept D, EMP M
  WHERE E.Dept = D.Dept #Check the correct department
      AND E.Sal > M.Sal #Check the salary condition
      AND D.Mgr = M.Name #Check the manager is the correct one
  )
THEN ROLLBACK;
```

The syntax of Starbust's rule definition is as described in Table 1. As we can see, rules have an unique name and each rule is associated with a single relation. The events are defined to be only database updates, but one rule can have several events defined on the target relation.

The same event can be used in several triggers, so one event can trigger different actions to be executed. For this not to produce an unwanted outcome, it is possible to establish the order in which different triggers must be executed, by using the PRECEDES and FOLLOWS declarations. The order defined by this operators is partial (not all triggers are comparable) and must be acyclic to avoid deadlocks.

```
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>];

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

Table 1: Starbust's rule definition syntax.

Example 2.2. *'If the average salary of employees gets over 100, reduce the salary of all employees by 10%.'*

In this case, the condition can be violated when a new employee is inserted, when an employee is deleted or when the salary is updated. Now, the action is not to rollback the operation, but to reduce the salary of every employee by 10%.

First, let's exemplify the cases in which the condition is violated. Imagine the following initial state of the table Emp:

Name	Sal
John	50
Mike	100
Sarah	120

The average salary is 90, so the condition is fullfilled.

- `INSERT INTO Emp VALUES('James', 200)`

  The average salary would be 117,5 and the condition is not fullfilled.

- `DELETE FROM Emp WHERE Name='John'`

  The average salary would be 110 and the condition is not fullfilled.

- `UPDATE Emp SET Sal=110 WHERE Name='John'`

  The average salary would be 110 and the condition is not fullfilled.

The trigger could be defined as:

```
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;
```

Note, nonetheless, that for the first example, we would get the following result:

Name	Sal
John	45
Mike	90
Sarah	108
James	180

Here, the mean is 105.75, still bigger than 100. We will see how to solve this issue later.

2.1.1 Starbust Semantics

At this point, it is interesting to bring some definitions up to scene:

Definition 2.1. A **transaction** is a sequence of statements that is to be treated as an atomic unit of work for some aspect of the processing, i.e., a transaction either executes from beginning to end, or it does not execute at all.

Definition 2.2. A **statement** is a part of a transaction, which expresses an operation on the database.

Definition 2.3. En **event** (in a more precise way than before) is the occurrence of executing a statement, i.e., a request for executing an operation on the database.

Thus, rules are triggered by the execution of operations in statements. In Starbust, rules are **statement-level**, meaning they are executed once per statement, even for statements that trigger events on several tuples. In addition, the execution mode is **deffered**. This means that all rules triggered during a transaction are placed in what is called the **conflict set** (i.e. the set of triggered rules). When the transaction finishes, all rules are executed in triggering order or in the defined order, if there is one. Nonetheless, if the need for a rule executing during a transaction exists, we can use the PROCESS RULES declaration, which executes all rules in the conflict set.

The algorithm for executing all rules after the transaction finished or PROCESS RULES is called is described in Algorithm 1. As we can see, rule processing basically involves 3 stages:

1. **Activation**: the event in the rule requests the execution of an operation and this is detected by the system.

2. **Consideration**: the condition of the rule is evaluated.
3. **Execution**: if the condition is fullfilled, the action in the rule is executed.

```
while CS is not empty
  select R in CS with highest priority
  delete R from CS
  if R.condition is TRUE
    execute R.action
```

Algorithm 1: processRules(conflict set CS)

2.1.2 Correctness of rules

Definition 2.4. The **repeatability of the execution** is the property that ensures that the system behaves in the same way when it receives the same input transaction in the same database state, i.e., the results are deterministic.

As we have seen before, rule definitions specify a partial order for execution, but several rules may have highest priority at the moment of selection. To achieve repeatability, the system maintains a total order, based on the user-defined partial order and the timestamps of the creation of the rules.

Definition 2.5. The **termination of rule execution** is reached when an empty conflict set is obtained.

Note that the execution of the rules may trigger more rules, this could cause **nontermination**, if two rules call each other in a cycle. Thus, ensuring termination is one of the main problems of active-rule design.

Example 2.3. Return to the previous example. We saw how the first of the examples did not end up fullfilling the condition, but it is because we did not take into account that the rule would trigger itself because it updates the Salary of Emp. Thus, the insertion and subsequent execution of the rule triggered gave us:

Name	Sal
John	45
Mike	90
Sarah	108
James	180

As we have updated the salaries, the rule is triggered again. The condition would be fullfilled, $105.75>100$ and the salaries would be modified again, arriving to the table as:

Name	Sal
John	41.5
Mike	81
Sarah	97.2
James	162

As the salaries ahve been updated again, the rule is triggered once more. Now, the mean is $95.425 < 100$, so the condition is not met and the actions are not executed. The rule has terminated.

In this case, termination is ensured because all values are decreased by 10%. This implies that the mean is also decreased by 10%. Thus, no matter how high the mean is at the beginning, at some point it will go below 100, because $0.9x < x$, $\forall x > 0$.

Remark 2.1. In general, guaranteeing termination is responsibility of the programmer is not an easy task.

Definition 2.6. A rule is **correct** (or possesses the correctness property) if it ensures repeatability and termination, taking into account the rest of factors in the database (other rules, domain of the attributes, structure of the tables,...).

2.1.3 State transitions and net effect

A transaction causes a state transition of the database, in the form of an addition, suppression of modification of one or more tuples of the database.

Before the transaction commits, the system stores two temporary **transition relations**, that contain the tuples affected by the transition. This tables are:

- INSERTED: for each event, it stores newly inserted tuples and the new form of tuples that have been modified.
- DELETED: for each event, it stores deleted tuples and the old form of tuples that have been modified.

Definition 2.7. The **net effect** of a transaction on a tuple is the composed effect of the transaction on the tuple, from the starting state to the end of the transaction.

Example 2.4. Some simple net effects:

- The sequence `INSERT` → `UPDATE` → `...` → `UPDATE` → `DELETE` on a newl tuple, has null effect: the tuple was not in the database at the beginning, and it is not there at the end.
- The sequence `INSERT` → `UPDATE` → `...` → `UPDATE` on a new tuple, has the same net effect as inserting the tuple with the values given in the last update.
- The sequence `UPDATE` → `...` → `UPDATE` → `DELETE` on an existing tuple, has the same net effect as just deleting it.

Remark 2.2. Rules consider the net effect of transactions between two database states, so each tuple appears at most once in each temporary table.

Example 2.5. Let's see the INSERTED and DELETED tables of our example. We start with the table

Name	Sal
John	50
Mike	100
Sarah	120

And perform `UPDATE Emp SET Sal=110 WHERE Name='John'.` If we accessed the temporary tables now, we would see:

INSERTED		DELETED	
Name	Sal	Name	Sal
John	110	John	50

Now, the rule is triggered because salary has been updated. The condition is met and the action is launched. If we accessed the temporary tables now, we would see:

INSERTED		DELETED	
Name	Sal	Name	Sal
John	99	John	50
Mike	90	Mike	100
Sarah	108	Sarah	120

Note how the first tuple in INSERTED shows only the net effect on the tuple.

With this definitions, we can give a more precise definition of rule triggering:

Definition 2.8. A rule is **triggered** if any of the transition relations corresponding to its triggering operations is not empty.

A great feature of rules is that it is possible to reference the transition tables, which can be very useful in many occasions.

Example 2.6. Now, imagine we want to add the rule '*If an employee is inserted with a salary greater than 100, add the employee to the table of hifh paid employees'.* This rule could be defined as:

```
CREATE RULE HighPaid ON Emp
WHEN
IF
  EXISTS (SELECT * FROM INSERTED WHERE Sal > 100)
THEN
  INSERT INTO HighPaidEmp
  (SELECT * FROM INSERTED WHERE Sal > 100)
FOLLOWS SalaryControl;
```

When we insert (James, 200), the tuple is inserted and the rules SalaryControl and HighPaid are triggered. Because we have defined HighPaid to follow SalaryControl, the latter would execute earlier. Now, SalaryControl, as we saw, would trigger more instances of the same rule, which would be all executed before HighPaid. At the end, as all employees would have been modified, all of them with a salary bigger than 100 would be added to the table HighPaidEmp because of this new rule. In this case, only James fullfills the condition.

2.1.4 More Starbust commands

```
1) DEACTIVATE RULE <rule-name> ON <table-name>
    Makes the specified rule not to be taken into account.

2) ACTIVATE RULE <rule-name> ON <table-name>
    Makes the specified rule to be taken into account.

3) DROP RULE <rule-name> ON <table-name>
    Deletes the specified rule.

4) CREATE RULESET <ruleset-name>
    Creates a ruleset, i.e., a set of related rules.

5) ALTER RULESET <ruleset-name> [ADDRULES <rule-names>] [DELRULES <rule-names>]
    Allows to add or delete rules to/from a ruleset.

6) DROP RULESET <ruleset-name>
    Deletes the specified ruleset (but not the rules).

7) PROCESS RULES
    Processes all active rules.

8) PROCESS RULESET <ruleset-name>
    Process a specified ruleset, if it is active.

9) PROCESS RULE <rule-name>
    Process a specified rule, if it is active.
```

Table 2: Starbust's rule commands.

2.2 Oracle

In Oracle, the term used is TRIGGER. Triggers in Oracle respond to modification operations (INSERT, DELETE, UPDATE) to a relation, just as in Starburst.

The triggers in Oracle can be defined for different granularities:

- **Tuple-level**: the rule is triggered once for each tuple concerned by the triggering event.
- **Statement-level**: the rule is triggered only once even if several tuples are involved.

Also, the execution mode of triggers in Oracle is **immediate**, meaning they are executed just after the event has ben requested, in contrast to Starburst, in which the execution mode is referred, as we saw. This allows for rules to be executed before, after or even instead of the operation of the triggering event. In 3, the definition syntax of triggers in Oracle is detailed. Some notes:

- The BEFORE or AFTER commands define when the rule should execute in relation to the events that trigger it.
- If FOR EACH ROW is written, then the trigger is a tuple-level trigger, and it is activated once for each tuple concerned.
  - This is useful if the code in the actions depends on data provided by the triggering statement or on the tuples affected.
  - The INSERTING, DELETING and UPDATING statements may be used in the action to check which triggering event has occurred.
  - OLD and NEW reference the old value of the tuple (if it is update or delete) and the new value of the tuple (if it is update or insert).
  - The condition consists of a simple predicate on the current tuple.
- If FOR EACH ROW is not written, then the trigger is a statement-level trigger, and it is activated once for each triggering statement even if several tuples are involved or no tuple is updated.
  - In this case, OLD and NEW are meaningless.
  - This is useful if the code of the actions does not depend on the data provided by the triggering statement nor the tuples affected.
  - It does not have a condition part[1].
  - It does not have the possibility to refer to intermediate relations as in Starburst[1].
- Oracle triggers can execute actions containing arbitrary PL/SQL[2] code (not just SQL as in Starburst).

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

where
  <trigger-event> := INSERT | DELETE | UPDATE [OF <column-names>]
  <references> := OLD as <old-tuple-name> | NEW as <new-tuple-name>
  <actions> := <PL/SQL block>
```

Table 3: Oracle's rule definition syntax.

Example 2.7. Row-level AFTER trigger.
Imagine we have two tables:

- `Inventory(Part, PartOnHand, ReorderPoint, ReorderQty)`
- `PendingOrders(Part, Qty, Date)`

We want to define a rule that whenever a PartOnHand is modified, and its new value is smaller than the ReorderPoint (i.e. we have less parts than the threshold to order more parts), we add a new record to PendingOrders as a new order for this part and the required quantity, if it is not already done. This can be done as follows:

```
CREATE TRIGGER Reorder
AFTER UPDATE OF PartOnHand ON Inventory
FOR EACH ROW
WHEN (New.PartOnHand < New.ReorderPoint)
  DECLARE NUMBER X;
  BEGIN
```

[1]It is not clear why Oracle engineers made it like this.
[2]PL/SQL extends SQL by adding the typical constructs of a programming language.

```
    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;
```

Let's apply the rule to see how it works. Let's say our table Inventory is:

Part	PartOnHand	ReorderPoint	ReorderQty
1	200	150	100
2	780	500	200
3	450	400	120

If we execute the following transaction on October 10, 2000:

```
UPDATE Inventory
SET PartOnHand = PartOnHand - 70
WHERE Part = 1;
```

Then the tuple `(1,100,2000-10-10)` would be inserted into PendingOrders.

The algorithm for executing rules in Oracle is shown in Algorithm 2. As we can see, statement-level triggers are executed before/after anything else, and row-level triggers are executed before/after each affected tuple is modified. Note that the executions needs to take into account the priority among triggers, but only those of the same granularity (row vs statement) and type (before vs after).

```
For each STATEMENT-LEVEL BEFORE trigger
  Execute trigger

For each row affected by the triggering statement
  - For each ROW-LEVEL BEFORE trigger
    Execute trigger
  - Execute the modification of the row
  - Check row-level constraints and assertions
  - For each ROW-LEVEL AFTER trigger
    Execute trigger

Check statement-level constraints and assertions

For each STATEMENT-LEVEL AFTER trigger
  Execute trigger
```

Algorithm 2: processRules

2.2.1 Oracle semantics

- The action part may activate other triggers. In that case, the execution of the current trigger is suspended and the others are considered using the same algorithm. There a maximum number of cascading triggerss, set at 32. When this 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 performed by triggers are rolled back. This means that Oracle supports partial rollback instead of transaction rollback.

2.2.2 Instead-of triggers

This is another type of Oracle trigger, in which the action is carried out inplace of the statement that produced the activating event. These triggers are typically used to update views and they need to be carefully used, because changing one action Y for an action X can sometimes have unexpected behaviors.

Example 2.8. An 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;
```

This trigger automatically updates the manager of a department when a new manager is inserted.

2.3 DB2

In DB2, every trigger monitors a single event, and are activated immediately, BEFORE or AFTER their event. They can be defined row-level or statement-level, as in Oracle. But in this case state-transition values can be accessed in both granularities:

- OLD and NEW refer to tuple granularity, as in Oracle.
- OLD_TABLE and NEW_TABLE refer to table granularity, like the DELETED and INSERTED in Starburst.

DB2's triggers cannot execute data definition nor transactional commands. They can raise errors which in turn can cause statement-level rollbacks.

The syntax is as in Table 4.

```
CREATE TRIGGER <trigger-name> {BEFORE|AFTER} <trigger-event>
ON <table-name>
[REFERENCING <references>]
FOR EACH {ROW|STATEMENT}
WHEN (<SQL-condition>)
<SQL-procedure-statements>;

where
  <trigger-event> := INSERT | DELETE | UPDATE [OF <column-names>]
  <references> := OLD as <old-tuple-name> | NEW as <new-tuple-name> |
      OLD_TABLE as <old-table-name> | NEW_TABLE as <new-table-name>
```

Table 4: DB2's rule definition syntax.

The processing is done as in Algorithm 3. Note that:

- Steps 1) and 6) are not required when S if part of an user transaction.
- If an error occurs during the chain processing of S, then the prior DB state is restored.
- IC refers to Integrity Constraints.

2.3.1 DB2 semantics

- **Before-triggers**: these are used to detect error conditions and to condition input values. They are executed entirely before the associated event and they cannot modify the DB (to avoid recursively activating more triggers).
- **After-triggers**: these are used to embed part of the application logic in the DB. The condition is evaluated and the action is possibly executed after the event occurs. The state of the DB prior to the event can be reconstructed from transition values.
- Several triggers can monitor the same event.
- In this case, the order is total and entirely based on the creation time of the triggers. Row-level and statement-level triggers are intertwined in the total order.

```
WHEN triggers ACTIVATE each other:
  IF a modification statement S in the action A of a trigger causes event E:
    1) SUSPEND execution of A, SAVE its data on a stack
    2) COMPUTE OLD and NEW relative to E
    3) EXECUTE BEFORE-triggers relative to E, update NEW
    4) APPLY NEW transition values to DB.
       FOR EACH IC violated by current state with action Aj:
      a) COMPUTE OLD and NEW relative to Aj
      b) EXECUTE BEFORE-triggers relative to Aj, update NEW
      c) APPLY NEW transition values to DB
      d) PUSH ALL AFTER-triggers relative to Aj into
         a queue of suspended triggers
    5) EXECUTE ALL AFTER-triggers relative to E
      IF ANY of them contains action Aj invoking other triggers:
        REPEAT RECURSIVELY
    6) POP from the stack the data for A, continue its evaluation
```

Algorithm 3: processRules

- If the action of a row-level trigger has several statements, they are all executed for one tuple before considering the next one.

Example 2.9. Imagine we have the following two tables:

Part		
PartNum	Supplier	Cost
1	Jones	150
2	Taylor	500
3	HDD	400
4	Jones	800

Distributor		
Name	City	State
Jones	Palo Alto	CA
Taylor	Minneapolis	MN
HDD	Atlanta	GA

And there is a referential integrity constraint that requires Part Suppliers to be also distributors, with HDD as a default Supplier:

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

Then, the following trigger is a row-level trigger that rollbacks 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');
```

2.4 SQL Server

In SQL Server, a single trigger can run multiple actions, and it can be fired by more than one event. Also, triggers can be attached to tables or views. SQL Server does not support BEFORE-triggers, but it supports AFTER-triggers (they can be defined using the word AFTER or FOR[3]) and INSTEAD OF-triggers.

The triggers can be fired with INSERT, UPDATE and DELETE statements.

The option WITH ENCRYPTION encrypts the text of the trigger in the syscomment table.

Finally, the option NOT FOR REPLICATION ensures that the trigger is not executed when a replication process modifies the table to which the trigger is attached.

The syntax is shown in Table 5.

- **INSTEAD OF-triggers**: are defined on a table or a view. Triggers defined on a view extend the types of updates that a view support by default. Only one per triggering action is allowed on a table or view. Note that views can be defined on other views, and each of them can have its own INSTEAD OF-triggers.

[3]FOR and WITH APPEND are used for backward compatibility, but will not be supported in the future.

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

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

Table 5: SQL Server's rule definition syntax.

- **AFTER-triggers**: are defined on a table. Modifications to views in which the table data is modified in response, will fire the AFTER-triggers of the table. More than one is allowed on a table. The **order of execution** can be defined using the *sp_settriggerorder* procedure. All other triggers applied to a table execute in random order.

2.4.1 SQL Server Semantics

- Both clases of triggers can be applied to a table.
- If both trigger classes and constraints are defined for a table, the INSTEAD OF-trigger fires first. Then, constraints are processed and finally AFTER-triggers are fired.
- If constraints are violated, INSTEAD OF-trigger's actions are rolled back.
- AFTER-triggers do not execute if constraints are violated or if some other event causes the table modification to fail.
- As stored procedures, triggers can be nested up to 32 levels deep and fired recursively.
- Two transition tables are available: INSERTED and DELETED, which are as in Starburst.
- The IF UPDATE(<column-name> clause determines whether an INSERT or UPDATE event ocurred to the column.
- The COLUMNS_UPDATE() clause returns a bit pattern indicating fhich of the tested columns were isnerted or updated.
- The @@ROWCOUNT function returns the number of rows affected by the previous Transact-SQL statement in the trigger.
- A trigger fires even if no rows are affected by the event. The RETURN command can be used to exit the trigger transparently when this happens.
- The RAISERROR command is used to display error messages.
- There are some Transact-SQL statements that are not allowd in triggers:
  - 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, then SET NOCOUNT ON must be included in the trigger code, disallowing the messages stating how many tuples were modified in each operation.

2.4.2 Limitations

- The INSTEAD OF DELETE and INSTEAD OF UPDATE triggers cannot be defined on tables that have a correspooonding ON DELETE or ON UPDATE cascading referential integrity defined.
- Triggers cannot be created on a temporary or system table, but they can be referenced inside other triggers.

2.4.3 Nested and Recursive triggers

SQL Server enables to enable or disable nested and recursive triggers:

- **Nested trigger option**: determines whether a trigger can be executed in cascade. There is a limit of 32 nested trigger operations. It can be set with *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: the neste trigger option must be set to true. This option can be set with *sp_dboption '<db-name>', 'recursive triggers', 'TRUE' / 'FALSE'.*

  Note that recursion can be **direct** if a trigger activates another instance of itself or **indirect** if the activation sequence is $T_1 \to T_2 \to T_1$. The recursive trigger option only copes with the direct recursion, the indirect kind is dealt with the nested trigger option.

2.4.4 Trigger management

Trigger management includes the task of altering, renaming, viewing, dropping and disabling triggers:

- Triggers can be modified with the ALTER TRIGGER statement, in which the new definition is provided.
- Triggers can be renamed with the *sp_rename* system stored procedure as

  sp_rename @objname = <old-name>, @newname = <new-name>

- Triggers can be viewed by querying system tables or by using the *sp_helptrigger* and *sp_helptext* system stored procedures as

  sp_helptrigger @tabname = <table-name>

  sp_helptext @objname = <trigger-name>

- Triggers can be deleted with the DROP TRIGGER statement.
- Triggers can be enable and disable using the ENABLE TRIGGER and DISABLE TRIGGER clauses of the ALTER TABLE statement.

Example 2.10. Let's work with a database with the following tables:

- `Books(TitleID, Title, Publisher, PubDate, Edition, Cost, QtySold)`
- `Orders(OrderId, CustomerId, Amount, OrderDate)`
- `BookOrders(OrderID, TitleId, Qty)`

Here, Books.QtySold is a derived attribute which keeps track of how many copies of the book has been sold. We can make this updates automatic with the use of the following trigger:

```sql
CREATE TRIGGER update_book_qtysold ON BookOrders
AFTER INSERT, UPDATE, DELETE AS
-- add if insertion
  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 i.titleID FROM INSERTED i)
  END
-- subtract if deletion
  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
```

- When there is an insertion in BookOrders, the trigger fires and adds the corresponding quantity.
- When there is a deletion, the trigger fires and subtracts the corresponding quantity.
- An update creates both tables, so we would add and subtract to cope with the modification.

3 Applications of Active Rules

Part II
Graph Databases

4 Introduction

5 Neo4j

Part III
Temporal Databases

Part IV
Spatial Databases

References

[1] Esteban Zimanyi. Infoh415 advanced databases. Lecture Notes.