03.Triggers

Rules that let DB to response to events computing actions. Enable applications to delegate some part of the behaviour to the DB. Rules tell what to do if something holds. Try to let DB express complex rules to have a good form of the data expressing rules inside the DB. They were developed to allow the DB to catch some events and respond to the events modifying the data inside. Each system express its triggers in its way.

Trigger Concept

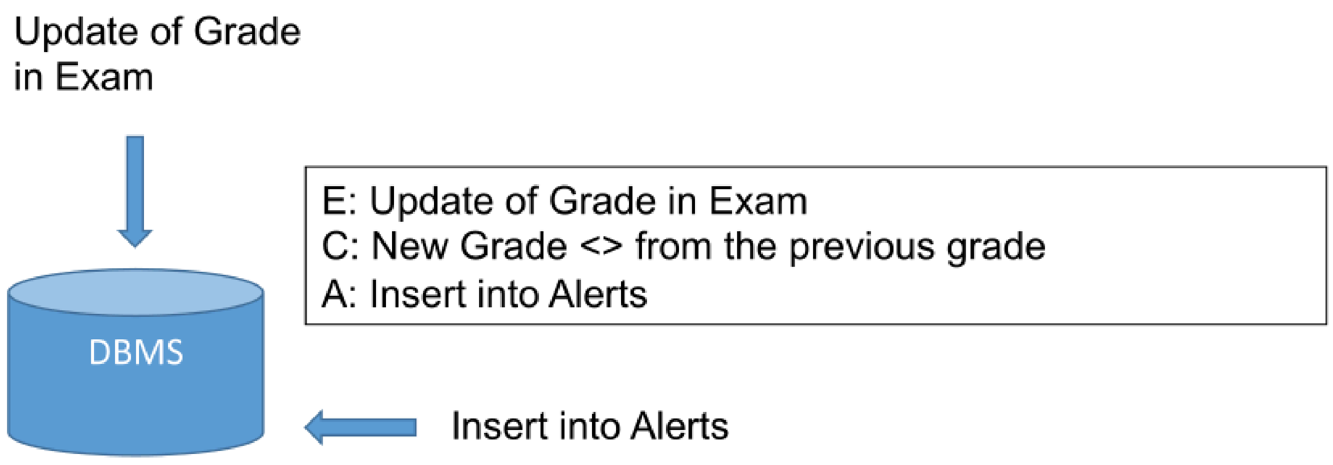
They are a triple of Event-Condition-Action(ECA) paradigm:

- whenever an event E occurs
- if a condition C is true
- then an action A is executed
  Reactive behaviour and implements complex constraints.
  Events are simple modification of data(addition, modification, deletion of a row)
- Event: Normally a modification of the database status: insert, delete, update
- Condition: A predicate that identifies those situations in which the execution of the trigger’s action is required
- Action: A generic update statement or a stored procedure. Usually database updates (insert – delete – update). Can include also error notifications

Active database – an example

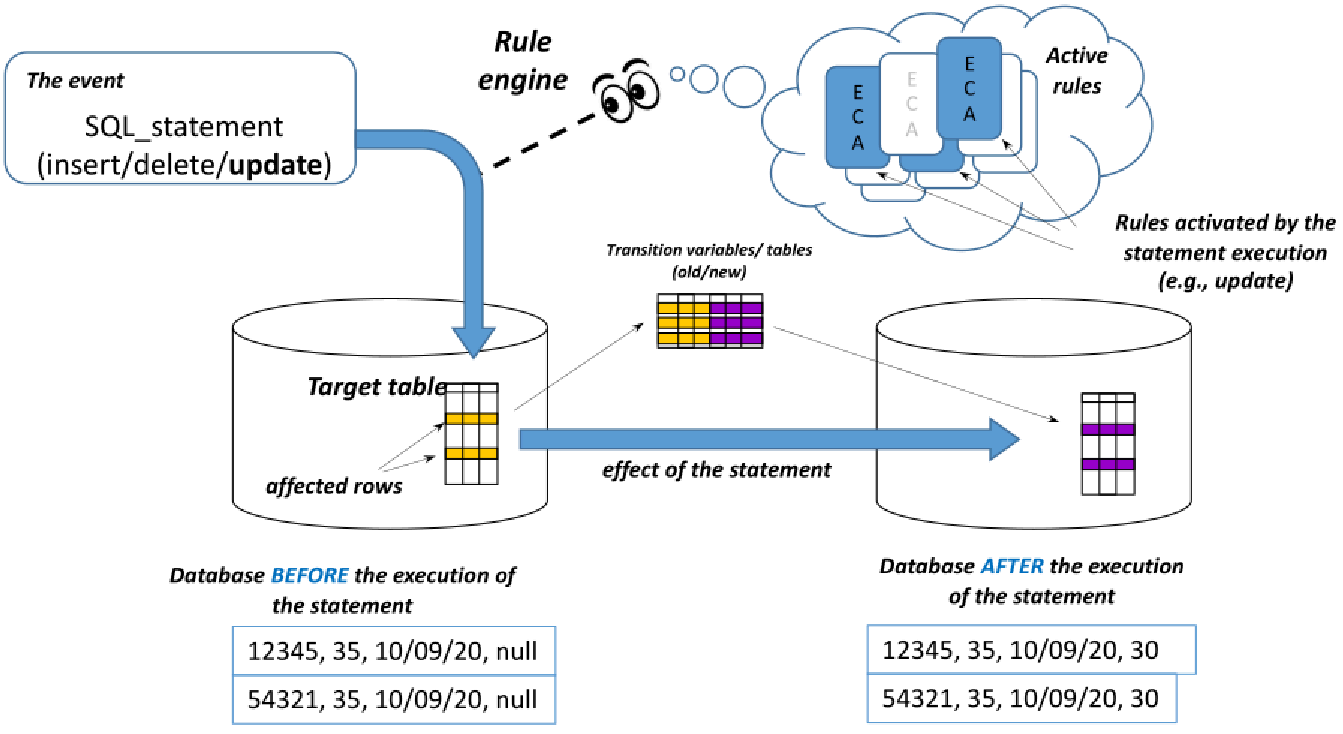
```
Student (StudID, Name, Email, …)
Exam(StudID, CourseID, Date, Grade,…)
Alerts(TS, StudID)
```

- Log an alert for the student if the Grade is updated



The big picture

```
UPDATE Exam SET Grade='30' WHERE StudID = '12345' OR
StudID = '54321' and CourseID='35';
```


The event
SQL_statement
(insert/delete/update)
Rule engine
E C A
E C A
E C A
Active rules
Rules activated by the statement execution (e.g., update)
Transition variables/ tables (old/new)
Target table
affected rows
effect of the statement
Database BEFORE the execution of the statement
12345, 35, 10/09/20, null
54321, 35, 10/09/20, null
Database AFTER the execution of the statement
12345, 35, 10/09/20, 30
54321, 35, 10/09/20, 30

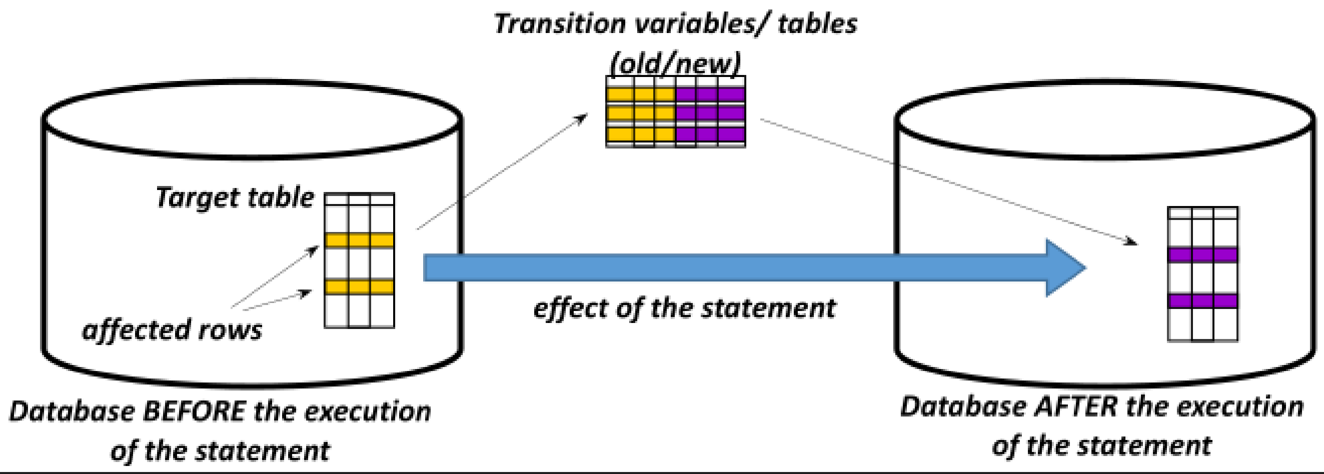
Triggers in SQL:1999, Syntax

```
  create trigger <TriggerName>
  { before | after }
E   { 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
C [ when <Condition> ]
A <SQLProceduralStatement>
```

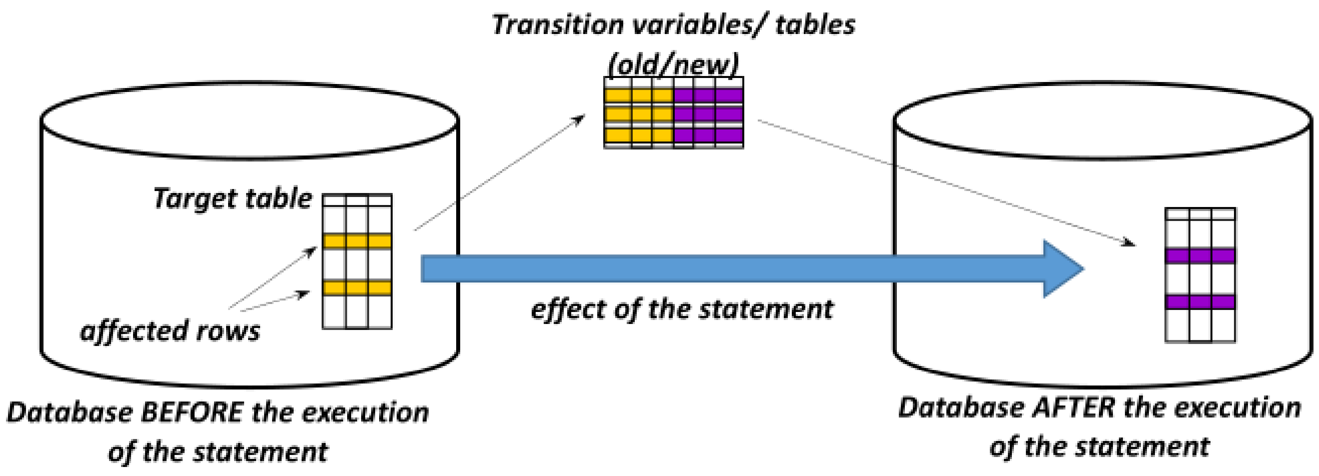
You need to specify when the trigger will happen, what is the event it is monitoring and the specific attributes it is looking for(you can look at old and new state and you can store that in a variable for each state as you are listening to events that modify the state of the DB. You can also specify if you want to respond to each row modification or if you want to respond to an update of multiple row.), you have the condition on when you fire the trigger and finally the action(sequence of SQL commands that modify the DB or do something else)

Before and After trigger

Before trigger is something that happens before the DB modification. It catches the update before it is executed and cancel it or modify it. We have access to the placeholders of new and old states and can modify them.

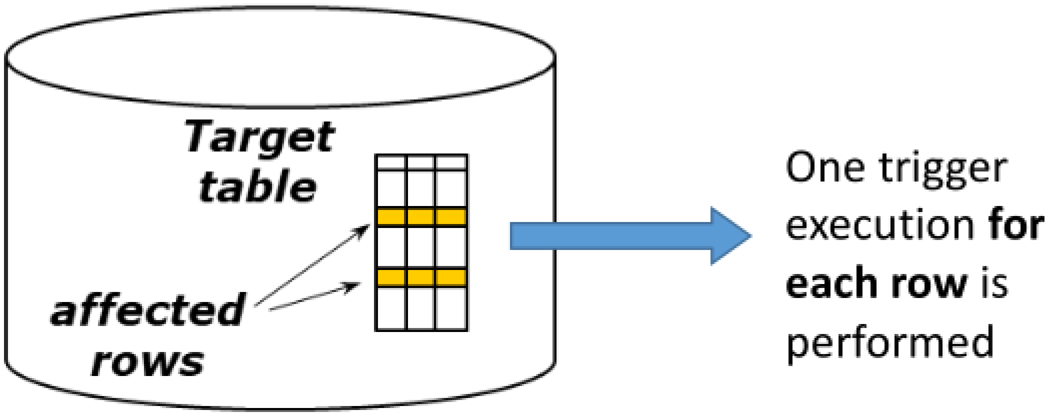

Transition variables/ tables
(old/new)
Target table
affected rows
effect of the statement
Database BEFORE the execution of the statement
Database AFTER the execution of the statement

After trigger execute after the completion of the update operation.

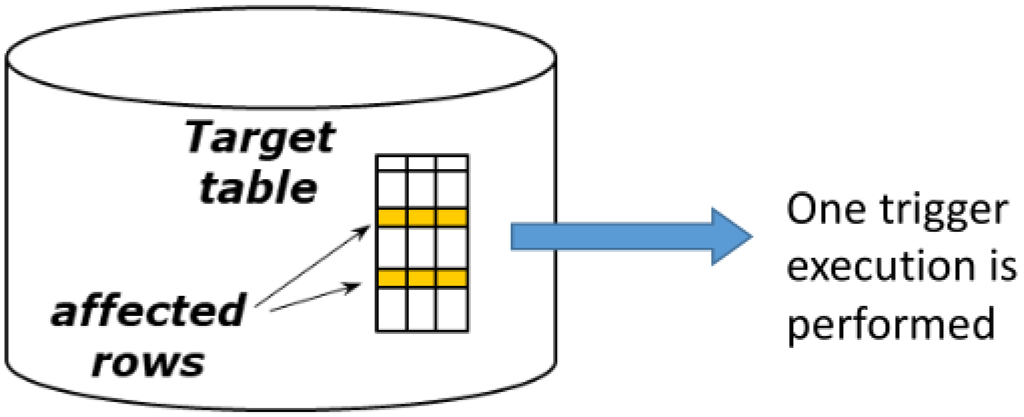

Transition variables/ tables (old/new)
Target table
affected rows
effect of the statement
Database BEFORE the execution of the statement
Database AFTER the execution of the statement

Granularity of events

Row level granularity: 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



Statement-level granularity (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.



Transition variables

At row level we have two variable called old and new and they refer to the single tuple before and after the update.
If we work at the Statement-level we need a table of variables to store the old state and the new state of the various tuple.
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
(old/new)
Transition tables
(old table/new table)

Handling all cases of UPDATE

Update of just the VALUE of a tuple in T1 (and ID is unchanged)

```
CREATE TRIGGER REPLIC_UPD
AFTER UPDATE OF VALUE ON T1
WHEN new.ID = old.ID
FOR EACH ROW
UPDATE T2 SET T2.VALUE = new.VALUE
        WHERE T2.ID = old.ID;
```

The previous trigger (correctly) won't fire when the ID of a tuple changes. However, a robust implementation must consider this case, too
In order to avoid too much clutter in the code, in these slides we only consider updates involving a
single, specific attribute. In the real world (and in exams, too), we would need to handle all cases

Conditional replication

Variant: Table T2 is a replication of table T1, but only the tuples whose value is >= 10 are replicated. Add condition in the WHEN clause to express more restrictions. All events affecting the replica must be treated.

```
Insertion operation:
CREATE TRIGGER CON_REPL_INS --new relevant tuple, replicate
AFTER INSERT ON T1
FOR EACH ROW
WHEN (new.VALUE >= 10)
INSERT INTO T2 VALUES (new.ID, new.VALUE);
```

```
Deletion:
CREATE TRIGGER Cond_REPL_DEL -- propagate deletion
AFTER DELETE ON T1
FOR EACH ROW
WHEN (old.VALUE >= 10)
DELETE FROM T2 WHERE T2.ID = old.ID;

Modification (only VALUE is modified and ID stays the same)
CREATE TRIGGER Cond_REPL_UPD_1 -- new relevant tuple, replicate
AFTER UPDATE OF VALUE ON T1
FOR EACH ROW
WHEN (new.ID = old.ID AND old.VALUE < 10 AND new.VALUE >= 10)
INSERT INTO T2 VALUES (new.ID, new.VALUE);

CREATE TRIGGER Cond_REPL_UPD_2 -- already replicated tuple changed, propagate
AFTER UPDATE OF VALUE ON T1
FOR EACH ROW
WHEN (new.ID = old.ID AND old.VALUE >= 10 AND new.VALUE >= 10 AND old.VALUE
!=
new.VALUE)
UPDATE T2 SET T2.VALUE = new.VALUE WHERE T2.ID = new.ID

CREATE TRIGGER Cond_REPL_UPD_3 -- replicated tuple no longer relevant: delete
AFTER UPDATE OF VALUE ON T1
FOR EACH ROW
WHEN (new.ID = old.ID AND old.VALUE >= 10 AND new.VALUE < 10)
DELETE FROM T2 WHERE T2.ID = new.ID;
```

BEFORE trigger

Example: prevent values to be updated to negative values. In such a case, they are set to 0
Statement:

```
UPDATE T1 SET T1.VALUE = -8 WHERE T1.ID = 5
```

Trigger:

```
CREATE TRIGGER NO_NEGATIVE_VALUES
BEFORE UPDATE of VALUE ON T1
FOR EACH ROW
WHEN (new.VALUE < 0)
SET new.VALUE=0; -- this "modifies the modification"
```

BEFORE vs AFTER

```
CREATE TRIGGER NO_NEGATIVE_VALUES
AFTER UPDATE of VALUE ON T1
```

```
FOR EACH ROW
WHEN (new.VALUE < 0)
UPDATE T1 SET VALUE = 0 WHERE ID=new.ID;
```

In some DB this is not allowed. Apparently an AFTER trigger does the same thing as the BEFORE trigger. But it is not equivalent: in the example, with the BEFORE trigger there is only 1 UPDATE statement, with the AFTER trigger there are 2 UPDATE statements. BEFORE triggers are more efficient if the goal is to "modify a modification". Some DBMSs (e.g., MySQL) disallow a trigger to update the table affected by the triggering event! You are listening an event on T1 and doing the action on T1 so the trigger could be fired recursively(this effect is disallowed by not powerful systems)

row vs. statement: DELETE event

Conditional replication with statement level trigger
Statements:

```
DELETE FROM T1 WHERE VALUE >= 5;
```

We can use aliases
Trigger:

```
CREATE TRIGGER ST_REPL_DEL
REFERENCING OLD TABLE AS OLD_T25
FOR EACH STATEMENT --all tuples considered at once
AFTER DELETE ON T1
DELETE FROM T2 WHERE T2.ID IN
(SELECT ID FROM OLD_T); -- no need to add where OLD_T.value >=10
```

With the for each statement we will have a single trigger to delete all the row in the statement.

row vs. statement: INSERT event

Statement:

```
INSERT INTO T1 (Id, Value) VALUES (4, 5), (5, 10), (6, 20);
```

Trigger:

```
CREATE TRIGGER ST_REPL_INS
AFTER INSERT ON T
REFERENCING NEW TABLE AS NEW_T
FOR EACH STATEMENT --all tuples considered at once
INSERT INTO T2
```

```
(SELECT ID, VALUE
FROM NEW_T WHERE NEW_T.VALUE >= 10);
```

row vs. statement: UPDATE event

Statement 1: double the value

```
UPDATE T1 SET value = 2 * value;
```

Statement 2: half the value

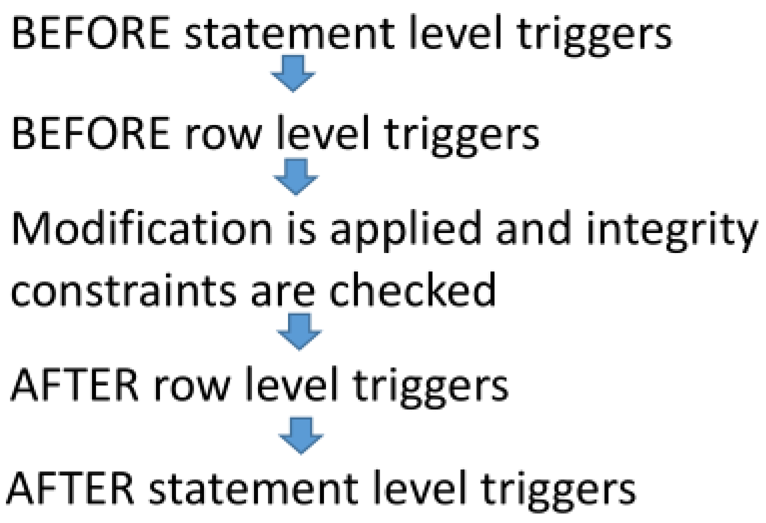
```
UPDATE T1 SET value = 0.5 * value;
```

Trigger:

```
CREATE TRIGGER REPLIC_INS
AFTER UPDATE ON T1
REFERENCING OLD TABLE AS OLD_T NEW TABLE AS NEW_T
FOR EACH STATEMENTIDVALUEIDVALUE
DELETE FROM T2 --delete all updated rows
WHERE T2.ID IN (SELECT ID FROM OLD_T);
INSERT INTO T2 --reinsert only relevant rows
(SELECT ID, VALUE FROM NEW_T
WHERE NEW_T.VALUE >= 10);
```

Multiple triggers on same event

Trigger Priority on same Event


BEFORE statement level triggers
BEFORE row level triggers
Modification is applied and integrity constraints are checked
AFTER row level triggers
AFTER statement level triggers

If several triggers are associated with the same event, SQL:1999 prescribes the execution sequence. 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) or based on the alphabetical order of the name. Some systems will look at the time the triggers are created and execute the oldest.

Cascading and recursive cascading

The action of a trigger can fire other triggers:

- Cascading: when the action of T1 triggers T2 (also called nesting)
- Recursive cascading: a statement S on table T starts a cascading of triggers that generates the same event S on T (also called looping). This can also happens on the same trigger firing itself
  This cascade operation will end?

Termination analysis

It is important to ensure that recursive cascading does not produce undesired effects
Termination: for any initial state and any sequence of modifications, a final state is always produced (infinite activation cycles are not possible)
The simplest check exploits 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 acyclic, the system is guaranteed to terminate
- If cycles exist, triggers may terminate or not
- Acyclicity is sufficient for termination, not necessary

We can have a three logic state: True, False and Unknown.
First concern is correctness.

Cascading triggers: a word of caution

When multiple triggers are activated by the same event the DBMS establishes a precedence criterion to decide which one to execute first. Different vendors implement different policies
Typically, the delayed triggers will be executed at a later moment
To fully understand what happens when those triggers are executed is far from trivial…

Why we use Trigger

Enables the DB with reacting behaviours and enabling complex query and we can use the triggers to do these complex computations instead of doing too complex query.
Another use is data replication and can be done smoothly with triggers.
Another application really useful that in some cases is materialised view maintenance: a view is a virtual table defined with a query stored in the database catalog and then used in queries as if

it were a normal table. The other queries still need to recalculate the view each time but transparently to you or each time you update the base tables you need to update the view. The last behaviour is easily produced using triggers.

View Materialization

When a view is mentioned in a SELECT query the query processor rewrites the query using the view definition, so that the actually executed query only uses the base tables of the view
When the queries to a view are more frequent than the updates on the base tables that change the view content, then view materialization can be an option
Storing the results of the query that defines the view in a table
Some systems support the CREATE MATERIALIZED VIEW command, which makes the view automatically materialized by the DBMS
An alternative is to implement the materialization by means of triggers

Incremental view maintenance

Not all modifications of Employee affect the view, and most modifications only affect a small part of the materialized data:

- Update of Idemp: no effect
- Insertion of an employee, deletion of an Employee, update of one salary attribute of employee: affects only one tuple in deptcost
- Update of dept attribute of one employee: affects only two tuples in deptcost
- Insertion and deletion of a department: affects only one tuple in deptcost
  When the effort to deal with the variation in the view is smaller than that of recomputing it from scratch, an incremental approach is preferable

Trigger design principles

1. Triggers should not be too complex as you need to understand their behaviour. You need to use trigger when the policy is general for the applications used in DBs.
2. Use triggers to guarantee that when a specific operation is performed, related actions are performed
3. Do not define triggers that duplicate features already built into the DBMS. For example, do not define triggers to reject bad data if you can do the same checking through declarative integrity constraints
4. Limit the size of triggers. If the logic for your trigger requires much more than 60 lines of code, it is better to include most of the code in a stored procedure and call the procedure from the trigger
5. Use triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or database application issues the statement.
6. Avoid recursive triggers if not absolutely necessary. Trigger may fire recursively until the DBMS runs out of memory.

7. Use triggers judiciously. They are executed for every user every time the event occurs on which the trigger is created