

Databases 2

Triggers

Active Databases

- Outline
 - Introduction: reactive behaviors
 - Trigger definition in SQL:1999
 - Execution semantics and properties of trigger-based systems
 - Examples
 - Evolution of triggers
 - Trigger definition in some commercial DBMSs

Definition and history

- TRIGGERS in DBMS:
 - “A set of actions that are automatically executed when an INSERT, UPDATE, or DELETE operation is performed on a specific table”
 - From “passive” to “active” behaviors
- 1975: Idea of “integrity constraints”
- Mid 1980-1990: research in constraints & triggers
- SQL-92: constraints
 - Key constraints, referential integrity, domain constraints
 - DECLARATIVE SPECIFICATION
- SQL-99: triggers/active rules
 - PROCEDURAL SPECIFICATION
 - Widely-varying support in products, with different execution semantics

The Trigger Concept

- Event-Condition-Action (ECA) paradigm
 - whenever an event E occurs
 - if a condition C is true
 - then an action A is executed
- Triggers are executed in addition to integrity constraints and allow one to check complex conditions
- They are compiled and stored in the DBMS similarly to stored procedures but are executed automatically based on the occurrence of events rather than invoked by the client

Event-Condition-Action

- 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

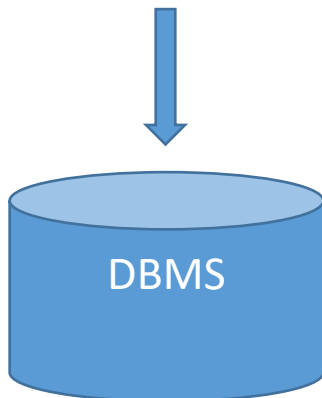
`Student (StudID, Name, Email, ...)`

`Exam(StudID, CourseID, Date, Grade,...)`

`Alerts (TS, StudID)`

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

Update of Grade
in Exam

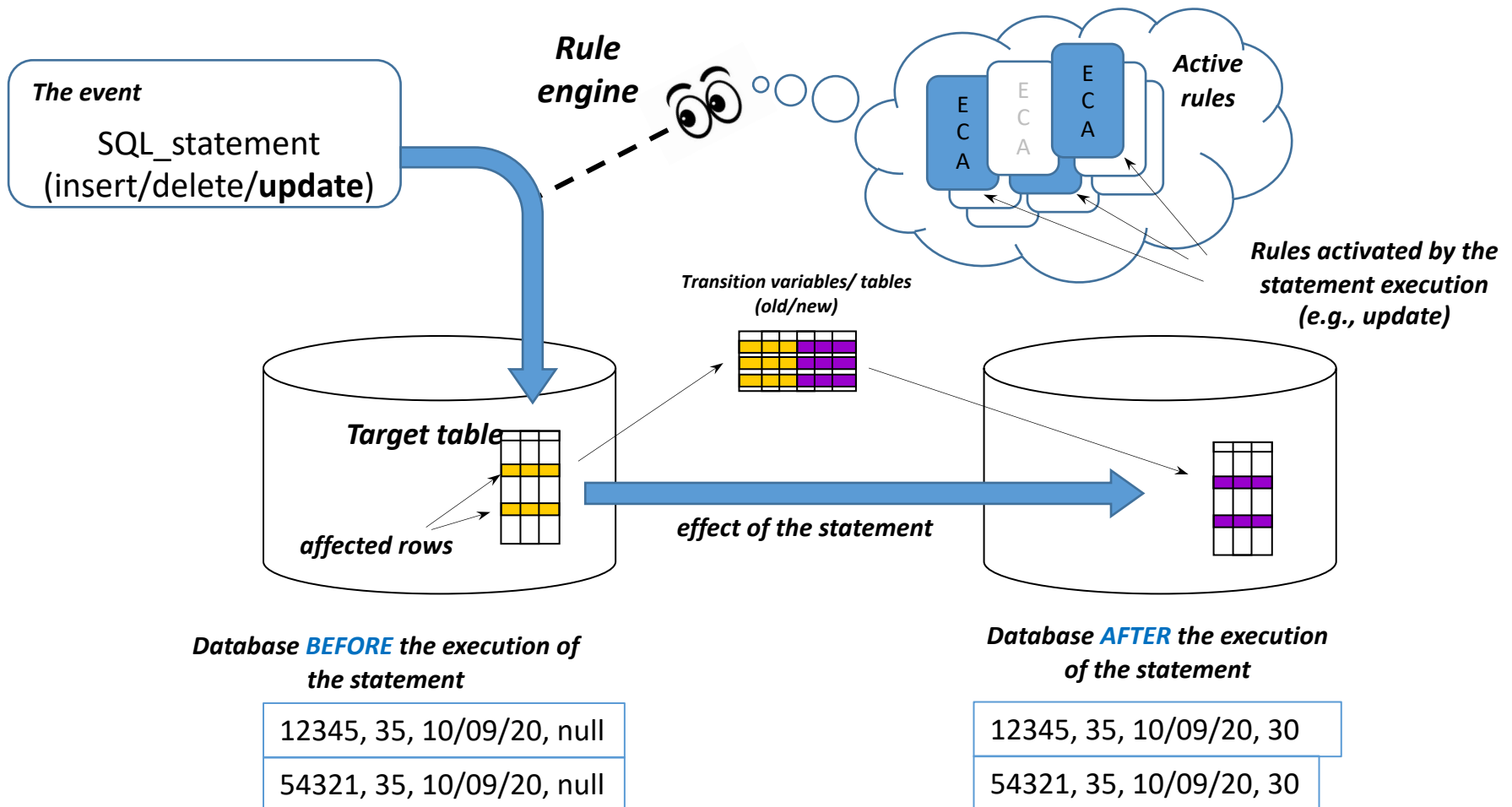


E: Update of Grade in Exam
C: New Grade \neq from the previous grade
A: Insert into Alerts

← Insert into Alerts

The big picture

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



Triggers in SQL:1999, Syntax

create trigger *<TriggerName>* triggers are objects in the DB, just like tables

{ before | after } --> when should the trigger intervene, normally we choose "after"

which kind of event should the trigger watch?

E { **insert** | **delete** | **update** [of <Column>] } **on** <Table>

```
referencing { [ old table [as] <OldTableAlias> ]
              [ new table [as] <NewTableAlias> ] |
```

just renaming old state
and new state

```
[ old [row] [as] <OldTupleName> ]
```

```
[ new [row] [as] <NewTupleName> ] }
```

[**for each** { **row** | **statement** }]

only once every statement (a statement can involve many statements)

for every

C [**when** *<Condition>*] involved row the trigger is activated

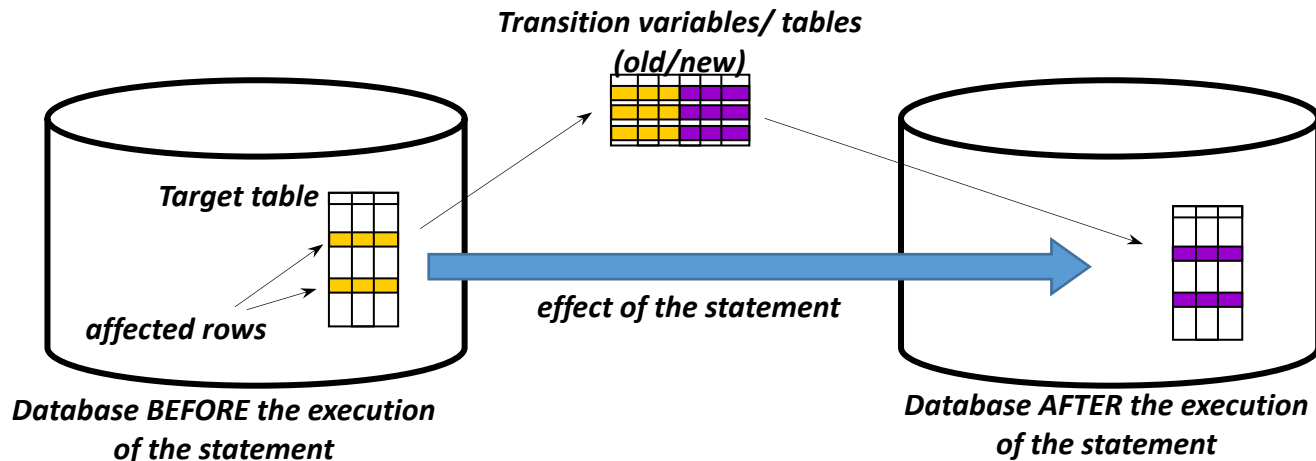
A <SQLProceduralStatement>

Execution modes: before or after

- **BEFORE**

- The action of the trigger is executed **before** the database status change (if the condition holds)
- Used to validate a modification before it takes place and possibly condition its effect
- Safeness constraint: before triggers cannot update the database directly, but can affect the transition variables in row-level granularity

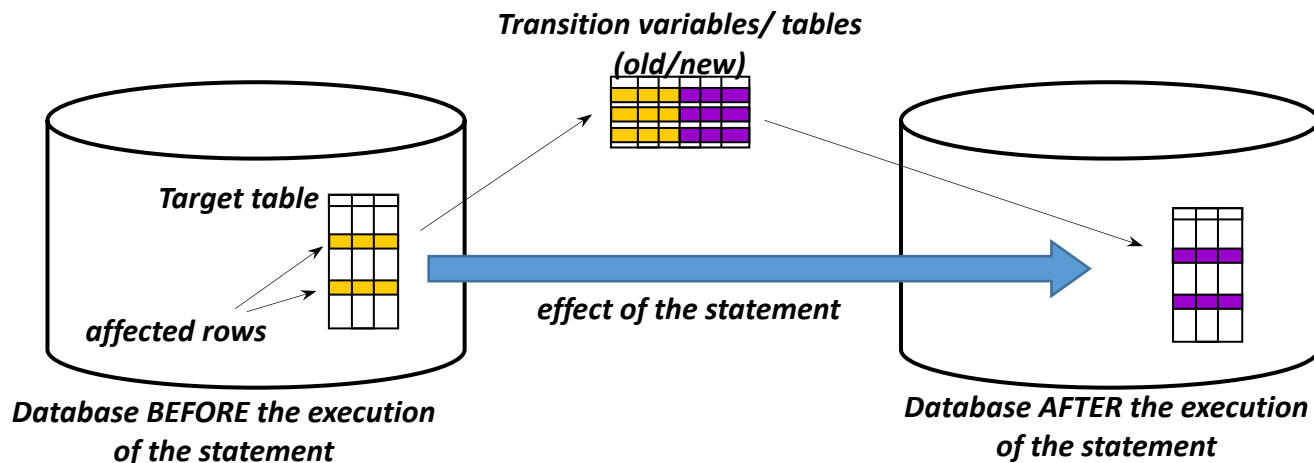
- **set new.t = <expr>**



Execution modes: before or after

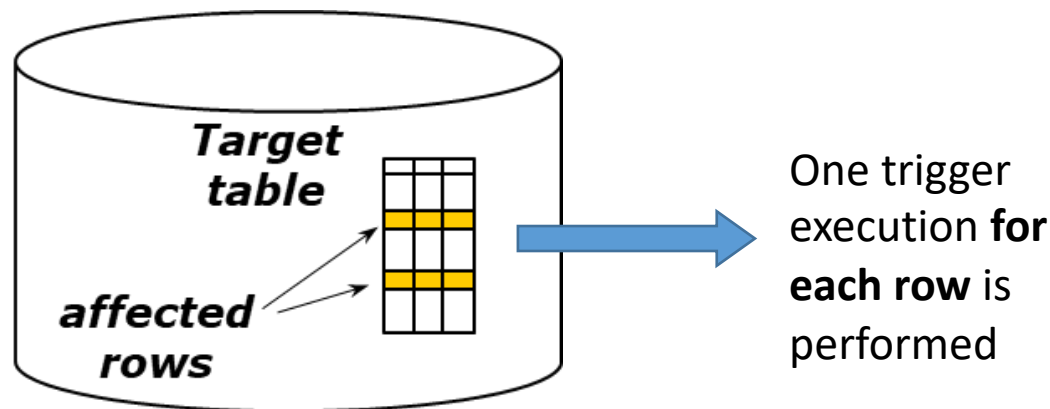
- **AFTER**

- The action of the trigger is executed **after** the modification of the database (if the condition holds)
- It is the most common mode, suitable for most applications



Granularity of events

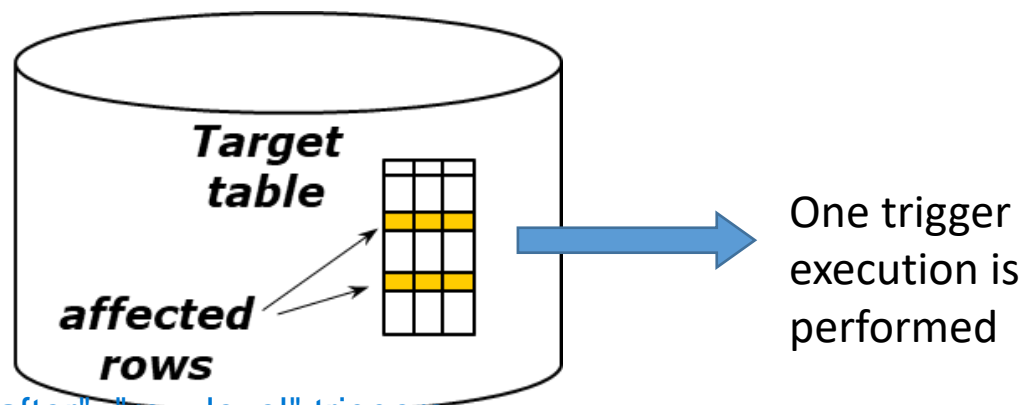
- **Row-level granularity** (for each row)
 - The trigger is considered and possibly executed **once for each tuple** affected by the activating statement
 - Writing row-level triggers is simpler, but can be less efficient



Granularity of events

- **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

Meaningful for efficiency reasons



Standard options used: "after", "row-level" triggers

Transition variables

- Special variables denoting the before and after state of the modification with a syntax that depends on the granularity
 - **Row-level**: tuple variables **old** and **new** represent the value respectively before and after the modification of the row (i.e., tuple) under consideration
 - **Statement-level**: table variables (**old table** and **new table**) contain respectively the old and the new values 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
(old/new)**

***Transition tables
(old table/new table)***

A simple example: data replication

- Table T2 is a replica of table T1: whenever an update occurs on T1, this is replicated on T2
- Replication is realized by means of triggers

replication is very common.

This kind of actions are implemented when some actions on the database must be hidden from the application

- Table T1

ID	VALUE
1	10
2	15



- Table T2

ID	VALUE
1	10
2	15

Insertion of a new tuple into T1

- Table T1:

ID	VALUE
1	10
2	15
3	20

- Table T2:

ID	VALUE
1	10
2	15
3	20

transition variable **new**

ID	VALUE
3	20

```
CREATE TRIGGER REPLIC_INS
AFTER INSERT ON T1
FOR EACH ROW
INSERT INTO T2 VALUES (new.ID, new.VALUE) ;
```

Deletion of a tuple from T1

- Table T1:

ID	VALUE
1	10
2	15
3	20

- Table T2:

ID	VALUE
1	10
2	15
3	20

transition variable **old**

ID	VALUE
2	15

```
CREATE TRIGGER REPLIC_DEL
AFTER DELETE ON T1
FOR EACH ROW
DELETE FROM T2 WHERE T2.ID = old.ID;
```


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

- Table T1:

ID	VALUE
1	10 5
2	15
3	20

- Table T2:

ID	VALUE
1	10 5
2	15
3	20

transition variables

old

ID	VALUE
1	10

new

ID	VALUE
1	5

```
CREATE TRIGGER REPLIC_UPD  
AFTER UPDATE OF VALUE ON T1
```

Condition -> WHEN new.ID = old.ID

```
FOR EACH ROW
```

```
UPDATE T2 SET T2.VALUE = new.VALUE
```

```
WHERE T2.ID = old.ID;
```

Handling all cases of UPDATE

- 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
- Homework: rewrite the previous trigger so that it also includes the case of an `ID` change

Conditional replication

- Variant: Table T2 is a replication of table T1, **but only the tuples whose value is ≥ 10 are replicated**
- 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  $\geq$  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  $\geq$  10)
DELETE FROM T2 WHERE T2.ID = old.ID;
```

Conditional replication

- 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"
```

old

ID	VALUE
5	6

new

ID	VALUE
5	-8

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;
```

Not allowed in some DBMSs



The final effect of this trigger is the same as the one that use "BEFORE"

- 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!

row vs. statement: DELETE event

- Conditional replication with statement level triggers

- Statement:

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

- Trigger:

```
CREATE TRIGGER ST_REPL_DEL
```

```
AFTER DELETE ON T1
```

```
REFERENCING OLD TABLE AS OLD_T
```

```
FOR EACH STATEMENT --all tuples considered at once
```

```
DELETE FROM T2 WHERE T2.ID IN
```

```
(SELECT ID FROM OLD_T); -- no need to add where OLD_T.value >=10
```

ID	VALUE
1	3
2	5
3	20

old table

ID	VALUE
2	5
3	20

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 T1
```

```
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);
```

ID	VALUE
1	3
2	5
3	20

new table

ID	VALUE
4	5
5	10
6	20

The effect is the same as the one obtained using row-level approach.

We have one insertion here, while in the row-level kind of syntax we would have many insertions. Possibly the statement approach is more efficient, but it is difficult to predict

row vs. statement: UPDATE event

- Statement 1: double the value

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

new_T / after stm 1

Initial DB state

T1 / old table

ID	VALUE
1	3
2	5
3	18
4	40

ID	VALUE
1	6
2	10
3	36
4	80

T2

ID	VALUE
2	10
3	36
4	80

- Statement 2: half the value

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

new_T / after stm 2

T2

ID	VALUE
3	18
4	40

ID	VALUE
1	1.5
2	2.5
3	9
4	20

T2

ID	VALUE
4	20

row vs. statement: UPDATE event

```
CREATE TRIGGER REPLIC_INS
AFTER UPDATE ON T1
REFERENCING OLD TABLE AS OLD_T NEW TABLE AS NEW_T
FOR EACH STATEMENT
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);
```

old T

ID	VALUE
1	3
2	5
3	18
4	40

T2

ID	VALUE
3	18
4	40

New_T / after stm 1

ID	VALUE
1	6
2	10
3	36
4	80

T2

ID	VALUE
2	10
3	36
4	80

New T / after stm 2

ID	VALUE
1	1.5
2	2.5
3	9
4	20

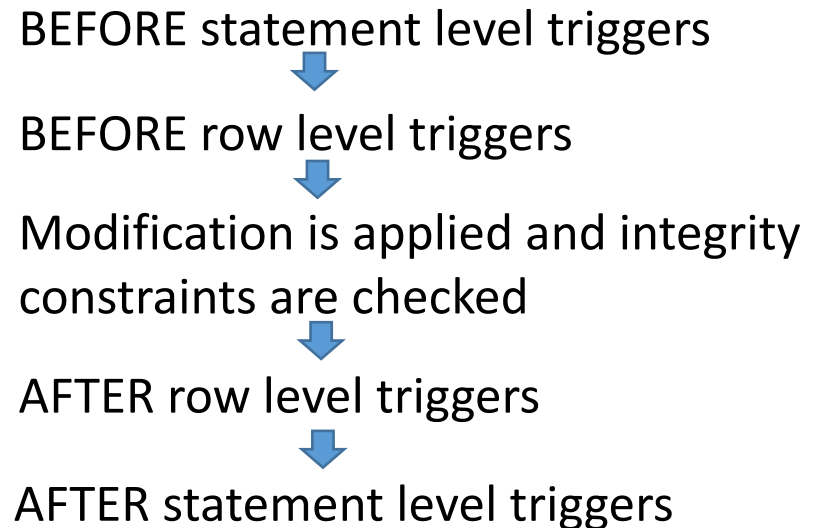
T2

ID	VALUE
4	20

Multiple triggers on same event

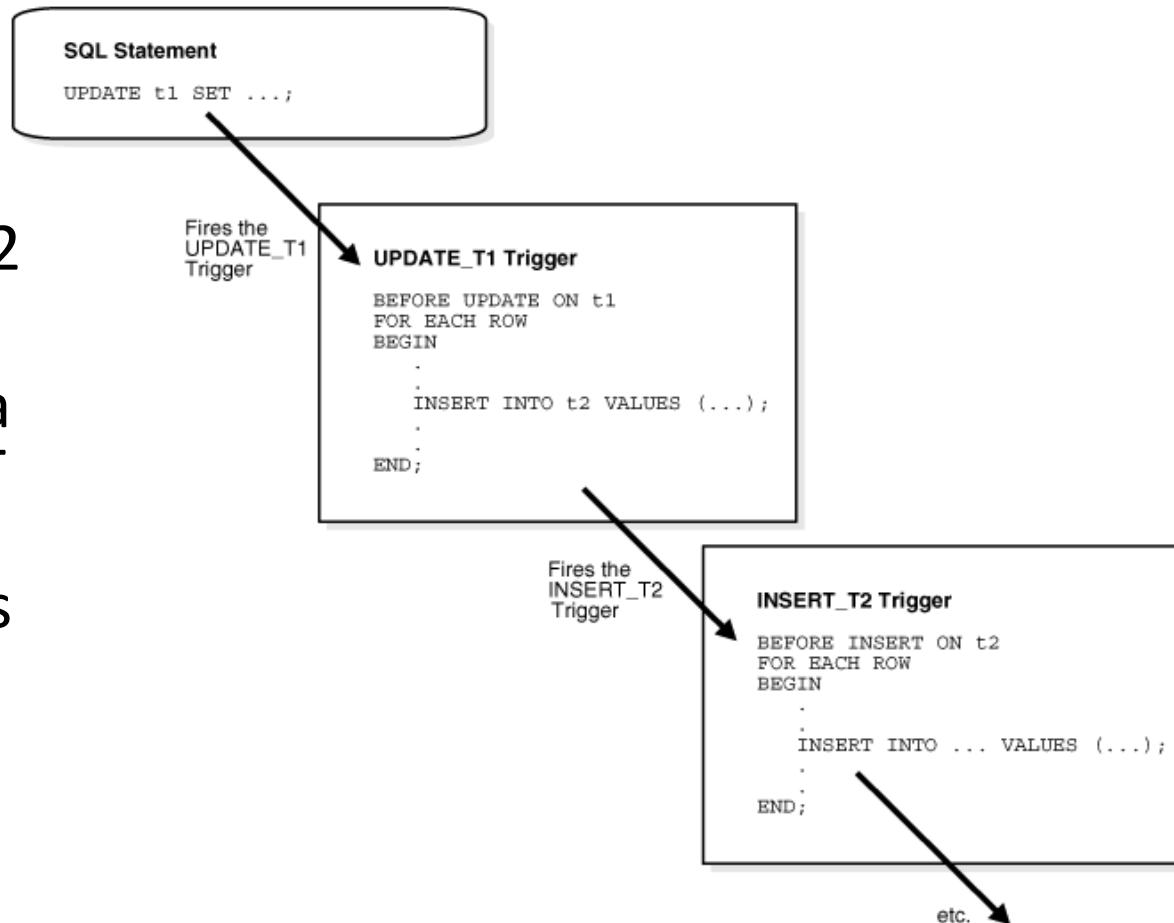
- If several triggers are associated with the same event, SQL:1999 prescribes the execution sequence
- If there are several triggers in the same category, the order of execution depends on the system implementation
 - e.g., based on the definition time (older triggers have higher priority) or based on the alphabetical order of the name

The order of execution in case different triggers are listening to the same event:



Cascading and recursive cascading

- The action of a trigger may cause another trigger to fire
- **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)



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**

Example

Employee (RegNum, Name, Salary, Contribution, DeptN, ...)

- T1:

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

listening to Salary and updating Contribution, so T1 will not be recursive by itself.

- T2:

```
create trigger CheckOverallBudgetThreshold
after update on Employee
for each statement
when (select sum(Salary+Contribution) from Employee) > 50000
update Employee
set Salary = 0.9 * Salary;
```

Here we listen and update salary, so could be recursive by itself and also with T1 (who listen's to Salary)

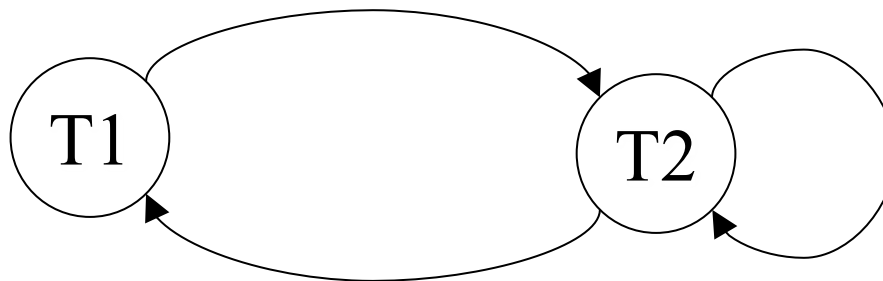
The reduction of the salary will reduce the contribution, which will reduce the salary+contribution sum, etc... until the >5000 condition will be no more satisfied and there will be a termination

Terminating cyclic triggering graph

- There are two cycles, but the system is terminating
- What if you invert the comparison in T2's condition?

- **when** (...) < 50000

--> in this case there would not be termination. The system will terminate until there is lack of memory or the maximum number of nesting is reached (in some systems)



How real systems work

- MySQL
 - A stored function or trigger cannot modify a table that is already being used (for reading or writing) by the statement that invoked the function or trigger
 - <https://dev.mysql.com/doc/refman/8.0/en/stored-program-restrictions.html>
- Postgres
 - If a trigger function executes SQL commands then these commands might fire triggers again. This is known as cascading triggers. There is no direct limitation on the number of cascade levels. It is possible for cascades to cause a recursive invocation of the same trigger; for example, an INSERT trigger might execute a command that inserts an additional row into the same table, causing the INSERT trigger to be fired again. It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios.
 - <https://www.postgresql.org/docs/9.1/trigger-definition.html>
- MS SQL Server
 - Triggers are nested when a trigger performs an action that initiates another trigger. These actions can initiate other triggers, and so on. Triggers can be nested up to 32 levels. An AFTER trigger does not call itself recursively unless the RECURSIVE_TRIGGERS database option is set.
 - <https://docs.microsoft.com/en-us/sql/relational-databases/triggers/create-nested-triggers?view=sql-server-ver15>
- Oracle
 - When a statement in a trigger body causes another trigger to be fired, the triggers are said to be cascading. Oracle Database allows up to 32 triggers to cascade at any one time. However, you can effectively limit the number of trigger cascades using the initialization parameter OPEN_CURSORS, because a cursor must be opened for every execution of a trigger.
 - https://docs.oracle.com/cd/B12037_01/appdev.101/b10795/adfns_tr.htm
- IBM Informix
 - Complicated...
 - <https://www.ibm.com/docs/en/informix-servers/12.10?topic=statement-cascading-triggers>

Example: book sales

- Consider the following relational schema:

BOOK(Isbn, Title, SoldCopies)

WRITING(Isbn, Name)

AUTHOR(Name, SoldCopies)

- Define a set of triggers for keeping SoldCopies in AUTHOR updated with respect to:
 - updates on SoldCopies in BOOK
 - insertion of new tuples in the WRITING relation

User updates BOOK table's SoldCopies, the triggers should automatically update the AUTHOR SoldCopies

Book sales: solution outline

BOOK(Isbn, Title, SoldCopies)
WRITING(Isbn, Name)
AUTHOR(Name, SoldCopies)

T1

T2

No cascading

- T1
 - Event: update of SoldCopies on BOOK
 - Condition: none
 - Action: update SoldCopies of AUTHOR for existing authors
- T2
 - Event: insert in WRITING
 - Condition: none
 - Action: update SoldCopies of AUTHOR for the new author of the book
- T3 ? This case is dismissed as not necessary
 - Event: insert in BOOK (or in AUTHOR)
 - Not necessary: to insert in WRITING a tuple of BOOK (and of AUTHOR) must already exist (due to referential integrity)

BOOK(Isbn, Title, SoldCopies)
WRITING(Isbn, Name)
AUTHOR(Name, SoldCopies)

Book update event

Assume for simplicity that `Isbn` cannot be updated

`create trigger UpdateSalesAfterNewSale`

`after update of SoldCopies on Book`

`for each row`

`update Author`

`set SoldCopies = SoldCopies`

`+ new.SoldCopies - old.SoldCopies`

`where Name in (select Name`

`from Writing`

`where Isbn = new.Isbn)`

WARNING!! Author.SoldCopies is NOT equal to Book.SoldCopies.

BOOK(Isbn, Title, SoldCopies)
WRITING(Isbn, Name)
AUTHOR(Name, SoldCopies)

Writing insert event

```
create trigger UpdateSalesAfterNewAuthorship
after insert on Writing
for each row
update Author
  set SoldCopies = SoldCopies +
    (select SoldCopies from Book where Isbn = new.Isbn)
where Name = new.Name
```

Example: sports competition

- Given the database

ATHLETE (ATHL_ID, **personal_record**, **qualified**)

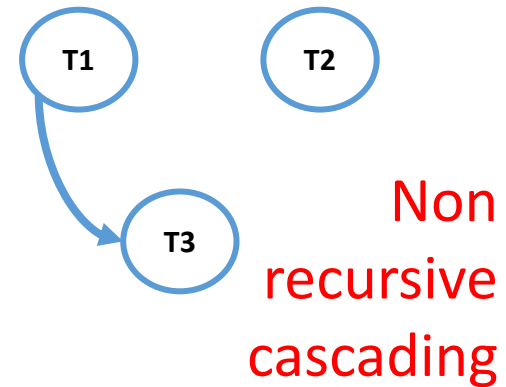
LOCAL_COMPETITION (COMP_ID, number_of_enrolled, **winner_id**)

RESULT (COMP_ID, ATHL_ID, time)

- At the end of a competition, an application inserts all the results of the athletes. Some triggers react to the insertion of the results of a competition, by:
 - Setting the **winner** of the competition
 - Updating the **personal record** if some athlete has improved her/his best time
 - Updating to true the **qualified** attribute of an athlete if s/he has won at least 10 local competitions

Sports competition: solution outline

- T1
 - Event: insert into RESULT
 - Condition: all results have been inserted into the database
 - Action: update LOCAL_COMPETITION set winner_id
- T2
 - Event: insert into RESULT
 - Condition: time < personal record
 - Action: update ATHLETE set personal record
- T3
 - Event: update of winner_id on LOCAL_COMPETITION
 - Condition: 10 competitions won
 - Action: update ATHLETE set qualified to true



```

ATHLETE (ATHL_ID, personal_record, qualified)
LOCAL_COMPETITION (COMP_ID, number_of_enrolled, winner_id)
RESULT (COMP_ID, ATHL_ID, time)

```

Trigger T1

- Event: insert into RESULT
- Condition: this is the last result & time < all other times
- Action: update LOCAL_COMPETITION set winner

```

create trigger UpdateWinner
after insert on result
for each row
-- fire only when all results have been inserted
WHEN (SELECT number_of_enrolled FROM local_competition
      WHERE COMP_ID = new.COMP_ID) =
      (SELECT count(*) FROM result WHERE COMP_ID = new.COMP_ID)
BEGIN
-- the competition winner is the one with the minimum time
UPDATE local_competition
SET winner_id = (select athl_id from result
                  where comp_id = new.comp_id and
                  time = (select min(time) from result
                          where comp_id = new.comp_id))
WHERE COMP_ID = new.COMP_ID
END;

```

```
ATHLETE (ATHL_ID, personal_record, qualified)  
LOCAL_COMPETITION (COMP_ID, number_of_enrolled, winner_id)  
RESULT (COMP_ID, ATHL_ID, time)
```

Trigger T2

- Event: insert into RESULT
- Condition: time < personal record or no personal record
- Action: update ATHLETE set personal record

```
create trigger UpdatePersonalRecord  
after insert on result  
for each row  
WHEN  
(SELECT personal_record FROM athlete  
WHERE ATHL_ID=new.ATHL_ID) IS NULL OR  
new.time < (SELECT personal_record FROM athlete  
WHERE ATHL_ID=new.ATHL_ID)  
BEGIN  
    UPDATE athlete  
    SET personal_record=new.time  
    WHERE ATHL_ID=new.ATHL_ID  
END
```


ATHLETE (ATHL_ID, **personal_record**, **qualified**)
LOCAL_COMPETITION (COMP_ID, number_of_enrolled, **winner_id**)
RESULT (COMP_ID, ATHL_ID, time)

Trigger T3

- Event: update of winner on COMPETITION
- Condition: 10 competitions won
- Action: update ATHLETE set qualified to true

```
create trigger updateQualification
after update of winner_id on local_competition
for each row
WHEN 10 = (SELECT count(*) FROM local_competition
           where winner_id = New.winner_id)
BEGIN
    UPDATE athlete
    SET qualified = true
    WHERE  ATHL_ID = NEW.winner_id
END
```

```
ATHLETE (ATHL_ID, personal_record, qualified)  
LOCAL_COMPETITION (COMP_ID, number_of_enrolled, winner_id)  
RESULT (COMP_ID, ATHL_ID, time)
```

Trigger T1 simplified

- Does this simpler version work?

```
create trigger UpdateWinner2  
after insert on result  
for each row
```

```
WHEN (SELECT min(time) FROM result  
      WHERE COMP_ID = new.COMP_ID) = new.time AND  
      (SELECT number_of_enrolled FROM competition  
       WHERE COMP_ID = new.COMP_ID) =  
      (SELECT count(*) FROM result  
       WHERE COMP_ID = new.COMP_ID)
```

```
BEGIN
```

```
    UPDATE local_competition  
    SET winner_id = new.ATHL_ID  
    WHERE COMP_ID = new.COMP_ID;
```

```
END
```

Inserting rows with multiple statements

- Multiple 1-row inserts

```
insert into result values (1,1,30);  
insert into result values (1,2,20); -- this is the winner  
insert into result values (1,3,40);  
insert into result values (1,4,50);
```

- The trigger is fired for each row
- When the trigger fires on the second row, only the first and second rows have been inserted
- Count (*) does not coincide with number_of_enrolled
- The real winner is missed

Inserting multiple rows in one statement

Postgres

```
insert into result values  
(1,1,30),  
(1,2,20),  
(1,3,40),  
(1,4,50);
```

- The trigger is fired for each row
- When the trigger fires on the second row all rows have been inserted
- Count (*) coincides with number_of_enrolled
- Min(time) identifies the winner

MySQL

```
insert into result values  
(1,1,30),  
(1,2,20),  
(1,3,40),  
(1,4,50);
```

- The trigger is fired for each row
- When the trigger fires on the second row only the first and second rows have been inserted
- Count (*) does not coincide with number_of_enrolled
- The real winner is missed

Hierarchy management

- Example: a hierarchy of products

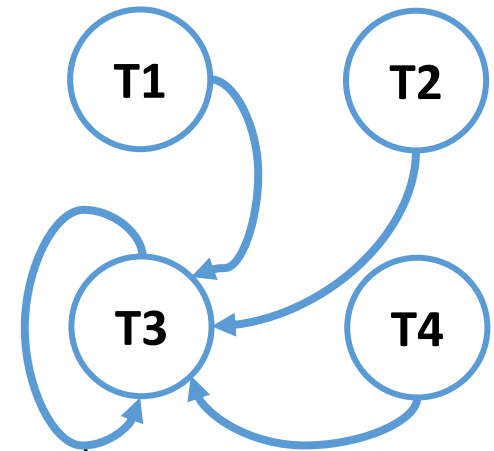
Product (ID, Name, SuperProduct, OwnWeight, TotalWeight)

(es: car wheels have car as superproduct)

- Each product is characterized by a super-product and by an own weight
- The total weight of a product is the sum of the weights of its sub-products (if they exist) and of its own weight
- Root products have:
 - SuperProduct = NULL
- Assume that users can insert and delete products and modify the superproduct of a product
- The total weight can be computed by a recursive view (with **recursive and hierarchical queries** in SQL:1999)
- The use of triggers to build and maintain the value of the total weight in the hierarchy is good design **ONLY when queries are much more frequent than updates**

Hierarchy management: solution outline

- T1
 - Event: insertion of a new product (user event)
 - Condition: none
 - Action: set total weight of product
- T2
 - Event: deletion of a product (user event)
 - Condition: superProduct not null
 - Action: decrease total weight of super product
- T3
 - Event: **update of total weight** of existing product (trigger event)
 - Condition: superProduct not null
 - Action: **update total weight** of super product
- T4
 - Event: update of superProduct of a product (user event)
 - Condition: superProduct different from previous super product
 - Action: decrease total weight of previous super product, increase total weight of new super product (if not null)
- Deletion of sub-products of a deleted product is managed by referential integrity ON DELETE CASCADE



**Recursive
cascading**

If the triggering graph is recursive we cannot guarantee the termination of the execution

Product (ID, Name, SuperProduct, OwnWeight, TotalWeight)

Trigger T1

- Event: insertion of a new product
- Condition: none
- Action: set total weight of product

```
CREATE TRIGGER product_AFTER_INSERT
AFTER INSERT ON product
FOR EACH ROW
BEGIN
UPDATE product SET totalweight = ownweight
WHERE ID = new.id;
END
```

Product (ID, Name, SuperProduct, OwnWeight, TotalWeight)

Trigger T2

- Event: deletion of a product
- Condition: superProduct not null
- Action: decrease total weight of super product

```
CREATE TRIGGER product_AFTER_DELETE
```

```
AFTER DELETE ON product
```

```
FOR EACH ROW
```

```
WHEN old.superproduct is not null
```



must be used when comparing something with null.

```
BEGIN
```

```
UPDATE product SET totalweight = totalweight - old.totalweight
```

```
WHERE ID = old.superproduct;
```

updating the weight of the superproduct



```
END
```

--> check if the product of I delete the superproduct is not null (it had a superproduct)

Trigger T3

- Event: update of total weight of existing product
- Condition: superProduct not null
- Action: update total weight of super product

```
CREATE TRIGGER product_AFTER_UPDATE_TOTALWEIGHT
AFTER UPDATE of totalweight ON product
FOR EACH ROW
WHEN new.superproduct is not null AND new.totalweight !=
old.totalweight
BEGIN
UPDATE product SET totalweight =  totalweight
                        + (new.totalweight - old.totalweight)
WHERE ID = new.superproduct;
END
```

Product (ID, Name, SuperProduct, OwnWeight, TotalWeight)

Trigger T4

- Event: update of superProduct of a product
- Condition: superProduct different from previous super product
- Action: decrease total weight of previous super product, increase total weight of new super product (if it exists)

```
CREATE TRIGGER product_AFTER_UPDATE_SUPER
AFTER UPDATE of superproduct ON product
FOR EACH ROW
WHEN new.superproduct is null or new.superproduct !=
old.superproduct      -- remember the three-valued logic...
                        We also have the "unknown" value.
BEGIN
UPDATE product SET totalweight = totalweight - old.totalweight
WHERE ID = old.superproduct;
UPDATE product SET totalweight = totalweight + new.totalweight
WHERE ID = new.superproduct;
END
```

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...

Example of what happens on Postgres

- Consider a simple table (`ttest`) with just one attribute (`x`) and always exactly one row
 - T1 halves `x` when `x >= 10` and T2 increases `x` by 40% when `x >= 6`


```
CREATE TRIGGER T1 AFTER UPDATE OF x ON ttest
FOR EACH ROW WHEN (NEW.x >= 10)
UPDATE ttest SET X = new.x / 2.0;
```

```
CREATE TRIGGER T2 AFTER UPDATE OF x ON ttest
FOR EACH ROW WHEN (NEW.x >= 6)
UPDATE ttest SET X = new.x * 1.4;
```

```
UPDATE ttest SET x = 12;
```

- Let us update x to the value 12 to start the dance...
- Assume its initial value was, e.g., 0

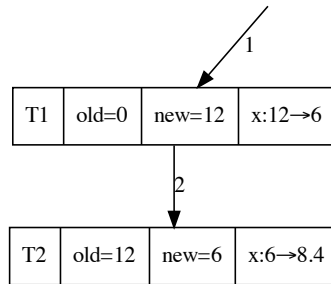
UPDATE ttest SET x = 12;



T1	old=0	new=12	x:12→6
----	-------	--------	--------

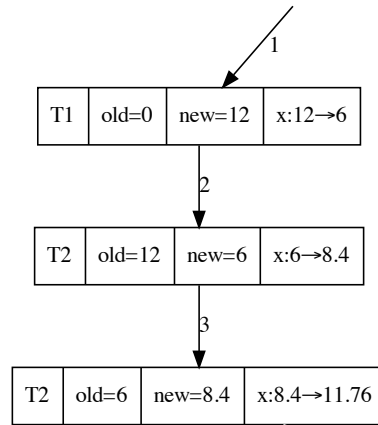
- The update activates both triggers, but T1 takes precedence because it precedes T2 alphabetically.
- The execution of T2 is delayed...
- The initial value of x was 0 (old), the update changed it to 12 (new) and T1 halves its value

UPDATE ttest SET x = 12;



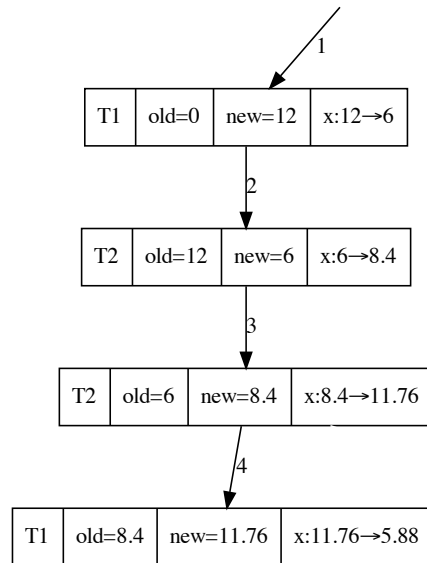
- The change made by T1 activates another instance of T2, which increases x by 40%, from 6 to 8.4

UPDATE ttest SET x = 12;

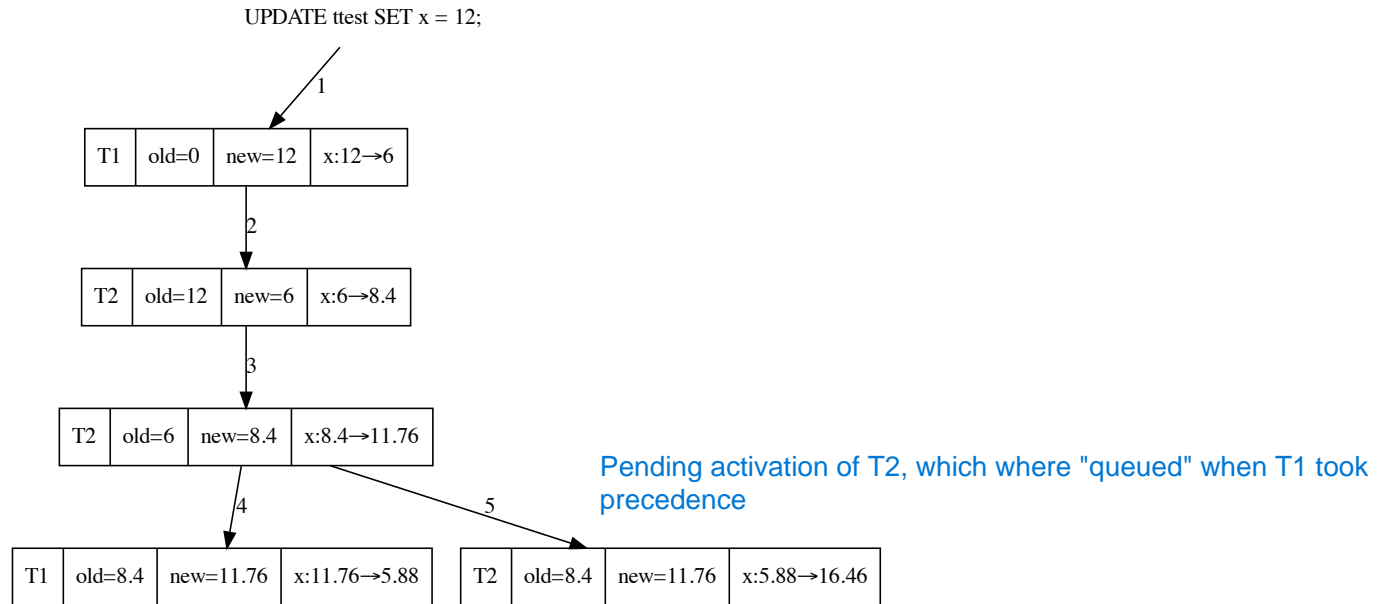


- Then another instance of T2 is executed, changing x from 8.4 to 11.76

UPDATE ttest SET x = 12;

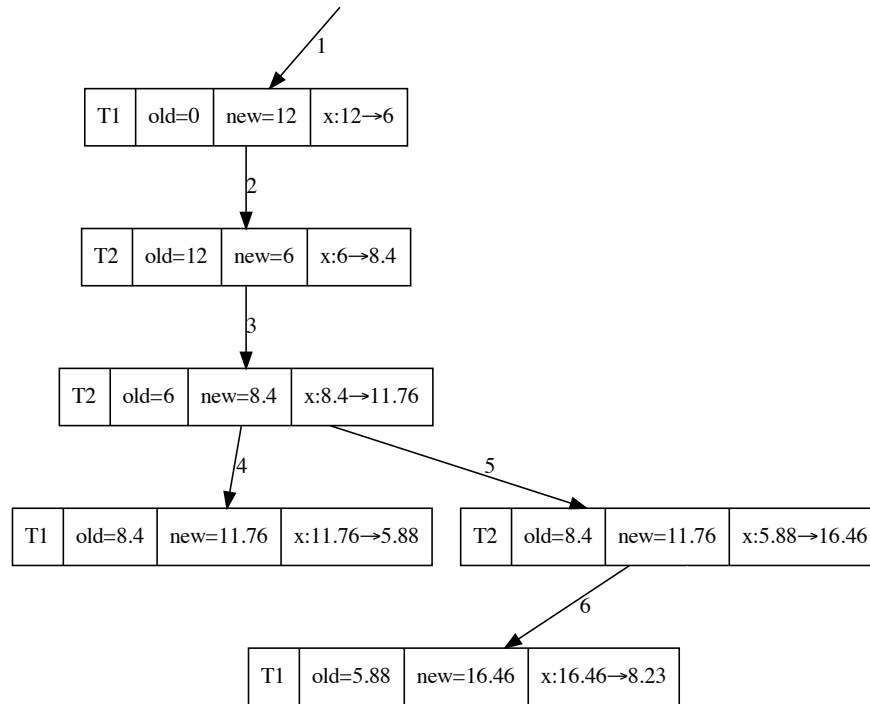


- Now both T1 and T2 are activated, but, again, T1 takes precedence, thereby halving the value of x, and T2 is delayed
- Are we done yet? Not quite...
- Now we can execute the (last) delayed instance of T2



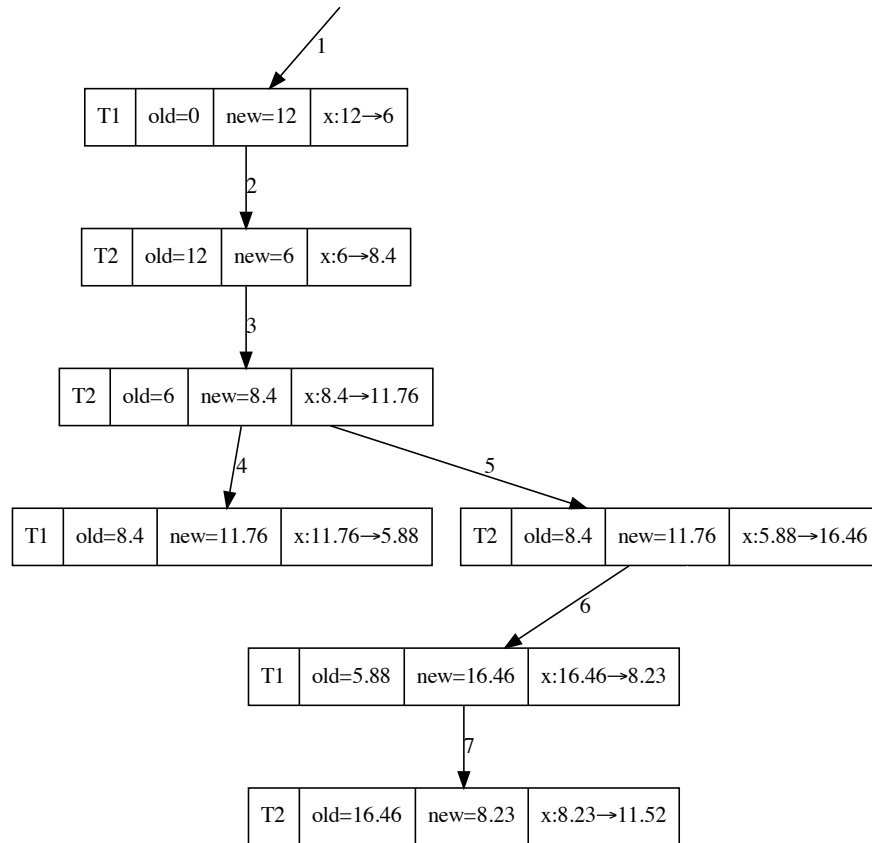
- At step 5, the current value of x is 5.88
- However, the “new” value of x that this instance of T2 sees is still 11.76
 - Indeed, it is executed after step 4, but was activated after step 3
- So, this execution changes x from 5.88 to 16.46
 - Not so obvious!

UPDATE ttest SET x = 12;



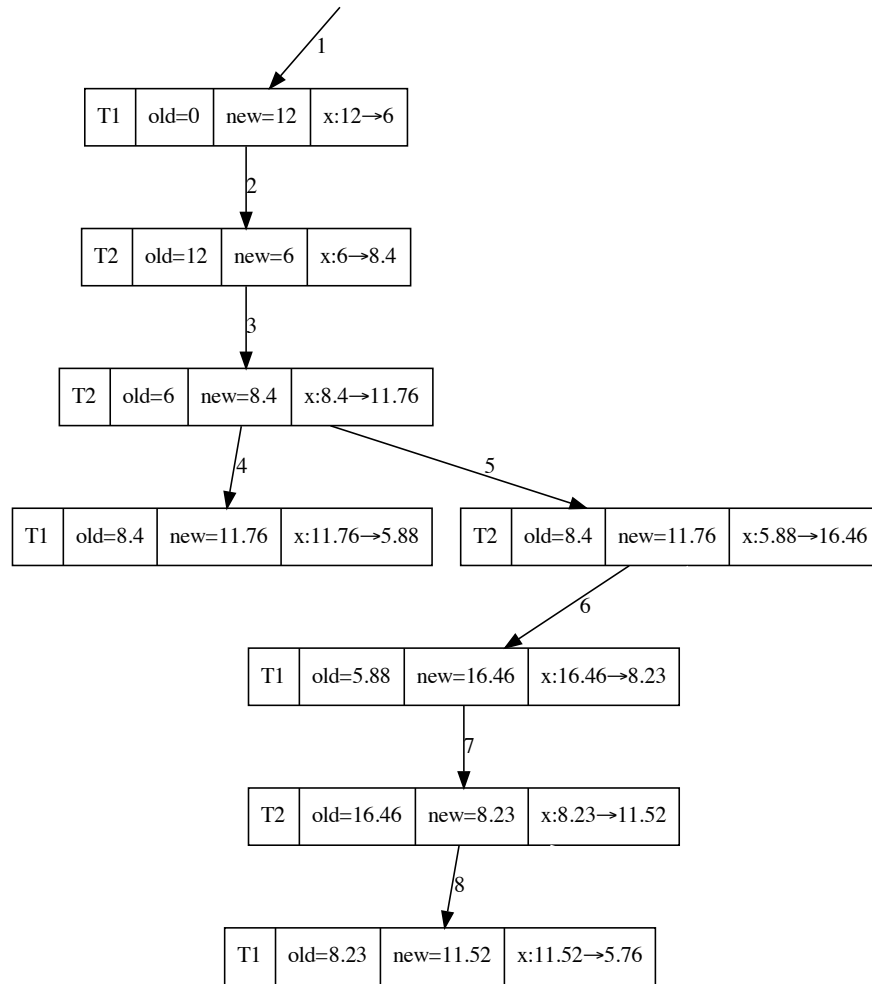
- At this point this chain of activations goes on

UPDATE ttest SET x = 12;



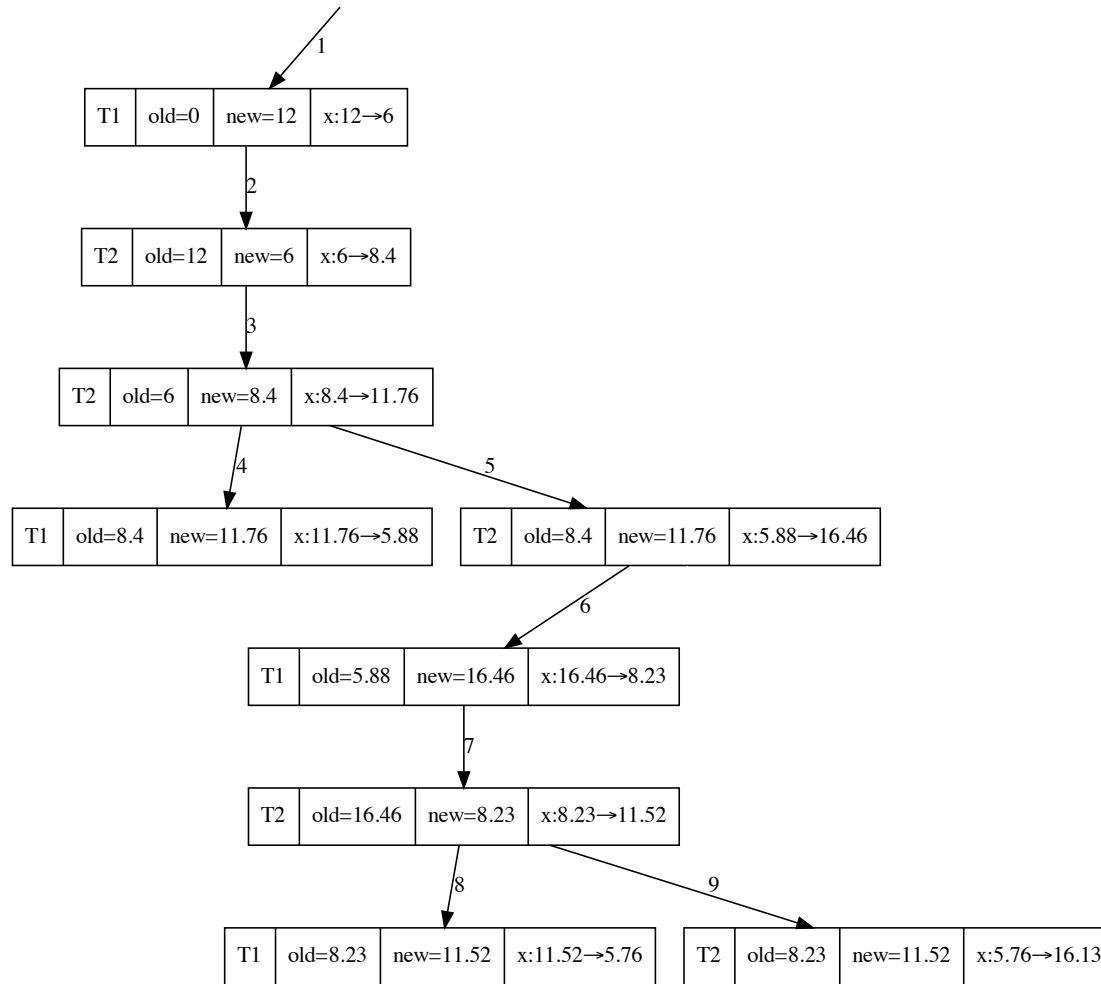
- ...and on

UPDATE ttest SET x = 12;



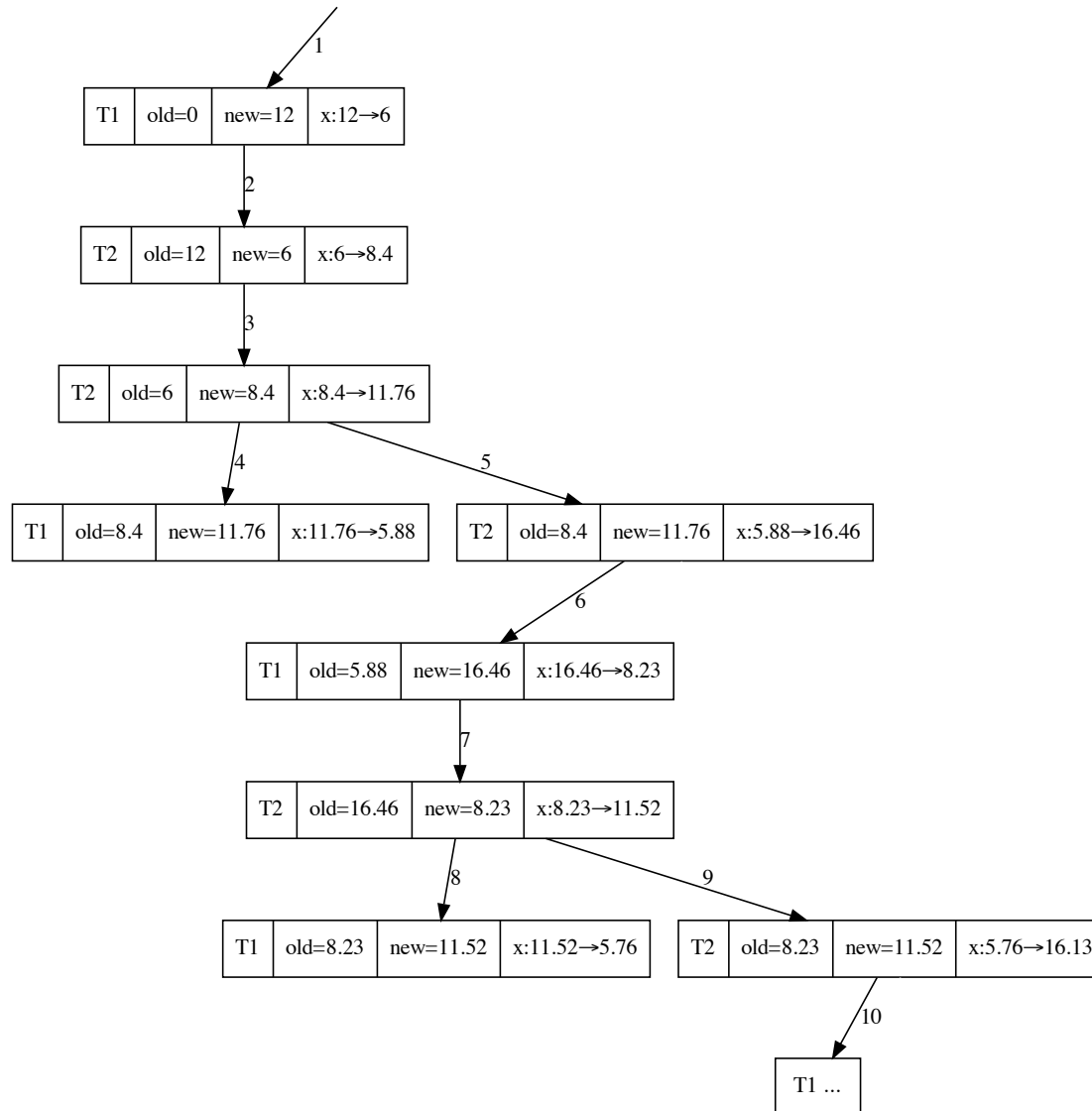
- ...and on

UPDATE ttest SET x = 12;

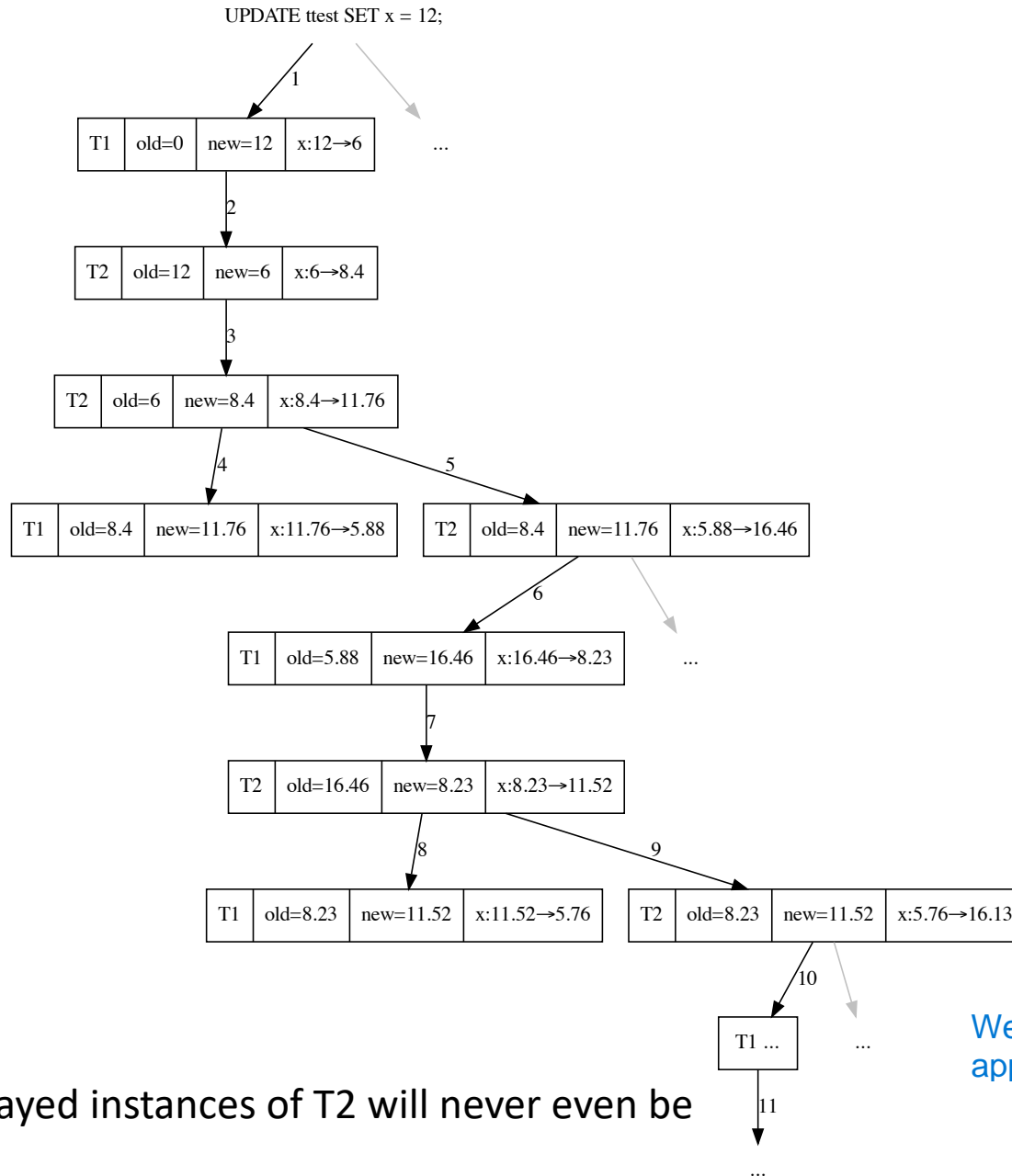


- ...and when the value of x becomes less than 6, we branch to execute a delayed instance of T2

UPDATE ttest SET x = 12;



- Here, the process never stops
- Eventually, the system will report an error



We get an infinite chain of applications

- Older, delayed instances of T2 will never even be executed

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
- Some DBMS vendors use triggers to implement internal services
- Examples:
 - Data replication
 - Integrity Constraint management not supported by declarative SQL primitives
 - Materialized view maintenance

Materialized Views

- 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
- A simple example: overall personnel cost of each department:

```
CREATE VIEW deptcost AS
```

```
SELECT d.DeptNum as dept, coalesce(sum(e.salary),0) as totCost  
FROM dept d LEFT JOIN emp e ON e.dept = d.DeptNum  
GROUP BY d.DeptNum;
```

- The view shows the personnel cost of every department

dept		emp			deptcost	
deptNum	Name	RegNum	Salary	Dept	dept	totCost
1	DEIB	1	100	1	1	264
2	DICA	2	80	2	2	168
		3	74	1		
		4	90	1		
		5	88	2		

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

dept		emp			deptcost	
deptNum	Name	Idemp	salary	dept	dept	totCost
1	DEIB	1	100	1	1	264
2	DICA	2	80	2	2	168
		3	74	1		
		4	90	1		
		5	88	2		

View maintenance: solution sketch

- T1
 - Event: insert on EMP
 - Condition: none
 - Action: increase totCost of deptCost
- T2
 - Event: delete on EMP
 - Condition: none
 - Action: decrease totCost of deptCost
- T3
 - Event: update of salary on EMP
 - Condition: dept is unchanged
 - Action: update totCost of deptCost
- T4
 - Event: update of dept of EMP
 - Condition: none
 - Action: increase totCost of new dept in deptCost, decrease totCost of old dept in deptCost
- T5
 - Event: insert on DEPT
 - Condition: none
 - Action: insert tuple and set totCost to 0 in DeptCost
- T6
 - Event: delete on DEPT
 - Condition: none
 - Action: delete on deptCost (can be managed with referential integrity)

No cascading: events on base tables, actions on materialized view table

Insertion /deletion of employees

```
create trigger Incremental_InsEmp
after insert on emp
for each row
update deptcost
  set totCost = totCost + new.salary
  where dept = new.dept;
```

```
create trigger Incremental_DelEmp
after delete on emp
for each row
update deptcost
  set totCost= totCost - old.salary
  where dept = old.dept;
```

row-level triggers

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

Salary update (and dept stays the same)

```
create trigger Incremental_SalaryUpdate
```

```
after update of salary on emp
```

```
when old.dept = new.dept
```

```
for each row
```

```
update deptcost
```

```
set totCost = totCost - old.salary + new.salary
```

```
where dept = new.dept;
```

Applies the “salary variation”...

idemp	salary	dept
1	100	1
2	80	2
3	74	1
4	90	1
5	88	2
6	75	2

80

dept	totCost
1	264
2	243

248

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

Update of dept (moving the employee)

```
create trigger Incremental_DeptChange
after update of dept on emp
for each row
begin
    update deptcost
        set totCost = totCost + new.salary
        where dept = new.dept;

    update deptcost
        set totCost = totCost - old.salary
        where dept = old.dept;
end;
```

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

idemp	salary	dept
1	100	1
2	80	2
3	74	1
4	90	1
5	88	2
6	80	2

1

dept	totCost
1	264
2	248

344

168

Creation/deletion of a department

```
create trigger Incremental_deptinsert
after insert on dept
for each row
begin
    insert into deptcost values (new.deptNum, 0) ;
end;

create trigger Incremental_deptdelete
after delete on dept
for each row
begin
    delete from deptcost where dept = old.deptNum;
end;
```

Triggers and integrity

- Integrity maintenance triggers (as all triggers) interact with referential integrity
- The previous incremental triggers assume referential integrity constraints between dept and emp

```
CREATE TABLE emp (  
    idemp int NOT NULL AUTO_INCREMENT,  
    salary int DEFAULT NULL,  
    dept int NOT NULL,  
    PRIMARY KEY (idemp) ,  
    CONSTRAINT 'empdept' FOREIGN KEY (dept)  
    REFERENCES dept (DeptNum) ON DELETE CASCADE)
```

- Deleting a department cascades the deletion on the affiliated employees
- Assigning an employee to a non existing department produces a constraint violation

Trigger design principles

- Use triggers to guarantee that when a specific operation is performed, related actions are performed
- 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
- 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
- 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.
- **Avoid recursive triggers if not absolutely necessary.** Trigger may fire recursively until the DBMS runs out of memory.
- Use triggers judiciously. They are executed for every user every time the event occurs on which the trigger is created
 - https://docs.oracle.com/cd/B12037_01/appdev.101/b10795/adfns_tr.htm

Evolution of active databases

- Execution mode (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 (e.g., Oracle)

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

Conclusions

- All major relational DBMS vendors have some support for triggers
- Most products support only a subset of the SQL-99 trigger standard
- Most do not adhere to some of the more subtle details of the execution model
- Some trigger implementations rely on proprietary programming languages
 - portability across different DBMSs is difficult
- Central management of semantics in DB, under the control of the DBMS (not replicated in all the applications)
- To guarantee properties of data that cannot be specified by means of integrity constraints
- Always document them, because their behavior is “hidden”