Triggers

Exercises

SQL:1999 Trigger Syntax

```
create trigger TriggerName
{before | after}
{ insert | delete | update [of Columns] } on
Table
[referencing
    {[old table [as] AliasOldTable]
     [new table [as] AliasNewTable] } |
    {[old [row] [as] NameTupleOld]
     [new [row] [as] NameTupleNew] }]
[for each { row | statement }]
[when Condition]
SQLCommands
```

Kinds of events

- BEFORE
  - The trigger is considered and possibly executed before the event (i.e., the database change)
  - Before triggers cannot change the database state; at most they can change (“condition”) the transition variables in row-level mode (set t.new=expr)
  - Normally this mode is used when one wants to check a modification before it takes place, and possibly make a change to the modification itself.
- AFTER
  - The trigger is considered and possibly executed after the event
  - It is the most common mode.

Granularity of events

- Statement-level mode (default mode, **`for each statement`** option)
  - The trigger is considered and possibly executed only once for each statement that activated it, independently of the number of modified tuples
  - Closer to the traditional approach of SQL statements, which normally are set-oriented
- Row-level mode (**`for each row`** option)
  - The trigger is considered and possibly executed once for each tuple modified by the statement
  - Writing of row-level triggers is simpler

Social

```
Highschooler(ID int, name text, grade int);
Friend(ID1 int, ID2 int);
Likes(ID1 int, ID2 int);
```

1 - Write one or more triggers to maintain symmetry in friend relationships. Specifically, if (A,B) is deleted from **Friend**, then (B,A) should be deleted too.
If (A,B) is inserted into Friend then (B,A) should be inserted too.
Don't worry about updates to the Friend table

2 - Write a trigger that automatically deletes students when they graduate, i.e., when their grade is updated to exceed 12. In addition, write a trigger so when a student is moved ahead one grade, then so are all of his or her friends.

3 - Write a trigger to enforce the following behavior: If A liked B but is updated to A liking C instead, and B and C were friends, make B and C no longer friends. Don't forget to delete the friendship in both directions, and make sure the trigger only runs when the "liked" (ID2) person is changed but the "liking" (ID1) person is not changed.

```
create trigger F1_Del
after delete on Friend
for each row
when exists (select * from Friend
                where ID1 = Old.ID2 and ID2 = Old.ID1)
begin
  delete from Friend
  where (ID1 = Old.ID2 and ID2 = Old.ID1);
end
```

```
create trigger F1_Insert
after insert on Friend
for each row
when not exists (select * from Friend
                where ID1 = New.ID2 and ID2 = New.ID1)
begin
  insert into Friend values (New.ID2, New.ID1);
end
```

```
create trigger Graduation
after update of grade on Highschooler
for each row
when new.grade > 12
begin
  delete from Highschooler
  where ID = New.ID;
end

create trigger Upgrade
after update of grade on Highschooler
for each row
when new.grade = Old.grade + 1
begin
  update Highschooler
  set grade = grade + 1
  where ID in (select ID2 from Friend
               where ID1 = New.ID);
end
```

```
create trigger NoLongerFriend
after update of ID2 on Likes
for each row
when Old.ID1 = New.ID1 and Old.ID2 <> New.ID2
begin
  delete from Friend
  where (Friend.ID1 = Old.ID2 and Friend.ID2 = New.ID2)
  or (Friend.ID1 = New.ID2 and Friend.ID2 = Old.ID2);
end
```

Bills

Bills

Consider the following relational schema that manages the telephone bills of a mobile phone company.

CUSTOMER (SSN, Name, Surname, PhoneNum, Plan)

PRICINGPLAN (Code, ConnectionFee, PricePerSecond)

PHONECALL (SSN, Date, Time, CalledNum, Seconds)

BILL (SSN, Month, Year, amount)

T1. Write a trigger that after each phone call updates the customer's bill.

T2. We make the assumption that the bills to be updated are always already present in the database. In order to do this, we can create another trigger that creates a bill with an amount of 0 for each registered customer at the beginning of each month (suppose we have the event *END_MONTH)*.

```
create trigger InitialBill
after END_MONTH
begin
    insert into BILL
    select SSN, sysdate().month, sysdate().year, 0
    from CUSTOMER
end
```

```
create trigger CallCharges
after insert of PHONECALL
for each row
begin
  update BILL B
  set Amount = Amount + ( select PP.ConnectionFee +
                  PP.PricePerSecond * new.Seconds
                  from PRICINGPLAN PP join CUSTOMER C
                     on C.Plan = PT.Code
                  where new.SSN = C.SSN )
  where B.SSN = new.SSN
     and B.Year = new.Date.year and B.Month = new.Date.month
end
```

T3. Write a trigger that at the end of each month discount the bills by 5 cents per call to direct users of the company (that is, to numbers of registered users in the table **CUSTOMER**) if the total monthly amount of the bill exceeds 100 €.

```
create trigger Offer
after END_MONTH
begin
    update BILL B
    set Amount = Amount – 0,05 * ( select count(*)
        from PHONECALL P
        where P.SSN = B.SSN
        and P.Date.month = ( sysdate() – 1 ).month
        and P.Date.year = ( sysdate() – 1 ).year
        and P.CalledNum in ( select PhoneNum from CUSTOMER) )
    where B.amount > 100 and B.year = (sysdate() - 1).year
        and B.month = (sysdate() - 1).month
end
```

Transactions

Consider the following relational schema:

TITLE (TitleCode, Name, Type)
TRANSACTION (TransCode, SellerCode, BuyerCode,
TitleCode, Quantity, Value, Date, Instant)
OPERATOR (Code, Name, Address, Availability)

Build a trigger system that keeps the value of Availability of Operator updated after insertion of tuples in Transaction, taking into account that for each transaction in which the operator sells, the amount of the transaction must be added to its availability and subtracted for purchases. Also, enter the operators whose availability falls below zero in a table that lists the operators "uncovered". Assuming that there is:

UNCOVERED (Code, Name, Address)

```
Create trigger TransferAvailability
after insert on TRANSACTION
for each row
begin
   update OPERATOR
   set Availability = Availability – new.Quantity * new.Value
   where Code = new.BuyerCode;
   update OPERATOR
   set Availability = Availability + new.Quantity * new.Value
   where Code = new.SellerCode;
end
```

```
Create trigger ReportUncovered
after update of Availability on OPERATOR
for each row
when new.Availability < 0 and old.Availability >= 0
begin
   insert into UNCOVERED values (new.Code, new.Name,
                                new.Address);
end
```

```
Create trigger RemoveUncovered
after update of Availability on OPERATOR
for each row
when old.Availability < 0 and new.Availability >= 0
begin
   delete from UNCOVERED where Code = new.Code
end
```

Championship

Consider the following relational schema:

MATCH (Day, HomeTeam, AwayTeam, HomeGoal, AwayGoal)
STANDING (Day, Team, Score)

Assuming that the first table is fed through entries and that the second is properly derived from the first, write the active rules that construct the ranking, giving 3 points to the teams that win, 1 point to those that tie and 0 points to those that lose.

```
create trigger HomeVictory
after insert on MATCH
when new.HomeGoal > new.AwayGoal
for each row
begin
  insert into STANDING S
  select new.Day, new.HomeTeam, S2.Score + 3
  from STANDING S2
  where S2.Team = new.HomeTeam and not exists
                   ( select * from STANDING where Day > S2.Day );
  insert into STANDING S
  select new.Day, new.AwayTeam, S2.Score
  from STANDING S2
  where S2.Team = new.AwayTeam and not exists
                   ( select * from STANDING where Day > S2.Day );
end
```

```
create trigger AwayVictory
after insert on MATCH
when new.HomeGoal < new.AwayGoal
for each row
begin
  insert into STANDING S
  select new.Day, new.HomeTeam, S2.Score
  from STANDING S2
  where S2.Team = new.HomeTeam and not exists
                  ( select * from STANDING where Day > S2.Day );
  insert into STANDING S
  select new.Day, new.AwayTeam, S2.Score + 3
  from STANDING S2
  where S2.Team = new.AwayTeam and not exists
                  ( select * from STANDING where Day > S2.Day );
end
```

```
create trigger Tie
after insert on MATCH
when new.HomeGoal = new.AwayGoal
for each row
begin
  insert into STANDING S
  select new.Day, new.HomeTeam, S2.Score + 1
  from STANDING S2
  where S2.Team = new.HomeTeam and not exists
                  ( select * from STANDING where Day > S2.Day );
  insert into STANDING S
  select new.Day, new.AwayTeam, S2.Score + 1
  from STANDING S2
  where S2.Team = new.AwayTeam and not exists
                  ( select * from STANDING where Day > S2.Day );
end
```

Volleyball

Consider the following relational schema for the european volleyball tournament:

PLAYER (PlayerId, Name, Team, Height, Birthday, PlayedMatches)

TEAM (Team, Coach, WonGames)

MATCH (MatchId, Date, Team1, Team2, WonSetsTeam1, WonSetsTeam2, Referee)

PLAYED (MatchId, PlayerId, Role, ScoredPoints)

1. Build a trigger that keeps the value of *WonGames* after insertions in GAME taking into account that *WonGames* is relative to the entire history of the team, not only to the current tournament, and that a team wins a game when he wins 3 sets.
2. Building also a trigger that keeps PlayedMatches of PLAYER updated after insertions in PLAYED.

```
create trigger IncrementWonGames
after insert on MATCH
for each row
begin
update TEAM
    set WonGames = WonGames + 1
    where
        new.WonSetsTeam1=3 and Team = new.Team1 or
        new.WonSetsTeam2=3 and Team = new.Team2
end
```

```
create trigger UpdatePlayedMatches
after insert on PLAYED
for each row
begin
    update PLAYER
        set PlayedMatches = PlayedMatches + 1
        where PlayerId = new.PlayerId
end
```

The “social” concert hall

A concert hall manages information about the shows using a set of row-level triggers. Visitors to the Web site can create an account and register a set of keywords matching their interests. When (a) a new show is inserted into the Website, with a set of keywords, registers users with a match with their set of keywords will receive an email. Some of them will buy a ticket for the event. In case of (b) show cancellation or (c) change of starting time, a notification is sent to users who bought a ticket for the affected show. Write only the triggers for the management of events (a,b,c). Assume that a function *send-mail*(*ReceiverEmail*, *Subject*, ... *OtherAttributes* ...) is available, which is invoked with all the parameters required for email creation. The database schema is:

VISITOR(VisId, Name, Email)   INTERESTS(VisId, Keyword)
SHOW(ShowId, Title, Date, StartTime)   DESCRIPTION(ShowId, Keyword)
TICKET(VisId, ShowId, Seats)

We assume that keywords are always inserted together with the show, and not updated

```
create rule NewShow
after insert into SHOW
for each row
 send-mail( ( select Email
              from ( VISITOR V join INTERESTS I on V.VisId = I.VisId )
                     join DESCRIPTION D on D.Keyword = I.Keyword
              where D.ShowId = new.ShowId ),
              “New Show”,
              new.Title,
              new.Date,
              new.StartTime )
```

```
create rule CanceledShow
after delete from SHOW
for each row
  send-mail( ( select Email
              from VISITOR V join TICKET T on V.VisId = T.VisId
              where T.ShowId = old.ShowId ),
              “Canceled Show” ,
              old.Title,
              old.Date,
              old.StartTime  )
```

```
create rule NewTime
after update of StartTime on SHOW
for each row
  send-mail( ( select Email
              from VISITOR V join TICKET T on V.VisId = T.VisId
              where T.ShowId = old.ShowId ),
              “Rescheduled Show”,
              old.Title,
              old.Date,
              new.StartTime )
```

Scholarship

Consider the following relational schema that manages the assignment of scholarships to students.

APPLICATION (StudentID, Date, State)

COURSE (CourseID, Title, Credits)

RANKING (StudentID, Average, Credits, Rank)

EXAM (CourseID, StudentID, Date, Grade)

We want to manage through a system of triggers the assignment of scholarships to students. The scholarships are awarded to students who apply and, at the date of the application, have taken exams for at least 50 credits, with an average score of at least 27/30.

- If the requirements are not met, the application is automatically rejected; otherwise accepted. In both cases, the value of the column *State* in **APPLICATION** is changed (initially it was NULL), respectively, with "rejected" or "accepted".
- In case of acceptance, the student is automatically assigned a position in the ranking, determined by the average of the grades; in case of equality of media, we consider first the greatest number of credits incurred at the date of application and finally the insertion order of the applications.
- If a student renounces the scholarship (*State* is changed to "dropout"), the ranking is updated.

Update the columns in **APPLICATION** e **RANKING** after insertion and deletions of applications.

```
Create trigger CheckApplication
after insert on APPLICATION
for each row
declare M, C number:
M := (    select avg( Grade ) from EXAM
          where StudentID = new.StudentID and Date <= new.Date )
C := (     select sum( Credits ) from  EXAM JOIN COURSE ON
          EXAM.CourseID = COURSE.CourseID
          where StudentID = new.StudentID and Date <= new.Date )
begin
if (M >= 27 and C >= 50)  then  (
        update APPLICATION set State=”accepted” where StudentID = new.StudentID;
        insert into RANKING values ( new.StudentID, M, C, NULL );   )
 else
        update APPLICATION set State=”rejected” where StudentID = new.StudentID;
end
```

```
Create trigger Dropout
after update of State  on APPLICATION
for each row
when new.State = “dropout”
begin
        delete * from RANKING where StudentID = new.StudentID;
end
```

```
Create trigger UpdateRanks1
after insert on RANKING
for each row
begin
POS:=select count(*)  // count the number of students with greater
                          // average grade or same average but more
credits
       from RANKING
      where(Average > new.Average) OR
            (Average = new.Average AND Credits > new.Credits) OR
            (Average = new.Average AND Credits = new.Credits);

update RANKING set Rank = Rank + 1 where Rank > POS;
// we move the following one down by one position (+1 in the ranking)

update RANKING set Rank = POS + 1 where StudentID =
new.StudentID;
// we attribute the «new» student the rank (pos + 1)
end
```

- The insertion order is implicitly considered, in fact when the average and the credits are both the same, the «older» applications are privileged.

- The rank is below the others with the same average and credits.

- POS is the number of preceding stundents in the ranking

```
Create trigger UpdateRanks2
after delete From RANKING
for each row
begin

     // update the position of the one that follows moving
     // them up by one

    update RANKING set Rank = Rank - 1
    where Rank > old.Rank;
end
```

Companies

Companies

Consider the following table, that keeps the stock shares owned by other companies:

OWNS (Company1, Company2, Percentage)

Consider, for simplicity, that tuples of the table OWNS are inserted from an empty table, which then no longer changes. Construct via trigger the relationship CONTROLS (a Company A controls a company B if A owns **directly or indirectly** more than 50% of B).

Assume that the control percentages are decimal numbers, and please note that the control situation is direct when a company owns more than 50% of the "controlled" or indirect when a company controls other companies that own percentages of B and the sum of the percentages owned and controlled by A exceeds 50%.

Schema of the auxiliary table:

CONTROLS (Controller, Controlled)

A can control B in two distinct ways:

- *directly*: it owns more than 50%
- *indirectly*: the sum of the percentage of direct and indirect (through other controlled companies) ownership is greater than 50% (even if none of these percentages exceed 50%)

The entries in OWNS can be managed with a trigger with statement granularity that translates the insertions in CONTROLS the ones that represent a direct control. The insertion in the table CONTROLS then trigger the search of indirect controls.

The ownerships with percentages > 50% are directly translated

```
create trigger DirectControl
after insert on OWNS
for each statement
do
      insert into CONTROLS select Company1, Company2
                              from new_table
                              where percentage > 50
```

Given a new control, it is necessary to propagate any indirect control

```
Create view IndirectPercentage( c1, c2, perc ) as
select C.Controller, O.Company2, sum( O.Percentage )
from CONTROLS C join OWNS O on C.Controlled = O.Company1
where ( C.Controller, O.Company2 ) not in ( select *
                                            from CONTROLS )
group by C.Controller, O.Company2
```

The view computes the percentage of indirect ownership not yet represented in the table CONTROLS. The trigger considers all percentages of indirect control *perc* (including those involved in the new tuple, because the trigger is *after*).

These percentages must be added to *potential* percentage of direct control (we can perform a **LEFT JOIN**, in order to consider all the indirect, and when there is no direct control for that pair of companies, the attribute will have a NULL value). The sum of perc and possible direct component is compared with 50.

```
create trigger IndirectContol
after insert on CONTROLS
for each row
do
    insert into CONTROLS  select c1, c2
                          from IndirectPercentage left join OWNS
                            on c1 = Company1 and c2 = Company2
                           where c1 = new.Controller  and
                             ( (Percentage is NULL and perc > 50) OR
               (Percentage + perc > 50) )
```

- *Note, that, in order to determinate an indirect control, it is not necessary a contribution from the table OWNS (and if it is present, is necessarily less than 50%, otherwise the control would have already been identified and inserted in the table as direct control).*

- *Instead, there must be a contribution by IndirectPercentage, so the LEFT JOIN.*

- *This trigger could conflict with the previous one, but the fact that in view we don't have tuples related to couples already in CONTROLS prevents any problem.*

A registration system assigns the classrooms of a conference center:

Registration (<u>Person</u>, <u>Session</u>)
Capacity (<u>Classroom</u>, NumSeatMax, Available)
Allocation (<u>Session</u>, Classroom)

Initially, 10 sessions (numbered from 1 to 10) are allocated to classrooms (numbered from 1 to 10) with a capacity of 25 people. There are also many other classrooms (sufficient to meet your needs).

1 - For each registration, you must ensure that the classroom scheduled for the session has a enough seats.

2 - When the number of seats are not enough you have to move the session to the smallest classroom among those available to house the participants, and make available the classroom previously allocated.

```
Create trigger updateAllocation
after insert on Registration
for each row
when
    select count (*) from Registration  where Session = new.Session
        >
    select NumSeatMax from Capacity C join Allocation A on C.Classroom = A.Classroom
    where Session = new.Session
begin
    update Capacity C join Allocation A on C.Classroom = A.Classroom
    set Available = 1 where Session = new.Session

    delete from Allocation  where Session = new.Session

    insert into Allocation
        (select new.Session, Classroom
        from Capacity
        where NumSeatMax =
                (select min(NumSeatMax) from Capacity
                where NumSeatMax >=
                    (select count (*)
                    from Registration
                    where Session = new.Session)
                and Available = 1)
        and Available = 1
        limit 1)
    update Capacity
    set Available = 0
    where Classroom =
        (select Classroom from Capacity
        where NumSeatMax =
            (select min(NumSeatMax) from Capacity
            where NumSeatMax >=
                (select count (*) from Registration
                where Session = new.Session)
            and Available = 1)
        and Available = 1
    Limit 1)
end
```

Bands

Consider the following schema describing a system for hiring rehearsal rooms to musical groups. Start and end time of reservations contain only hours (minutes are always “00”). Use of rooms can only happen if there is a corresponding reservation, **but can start later or end earlier**. All rooms open at 7:00 and close at 24:00.

User (SSN, Name, Email, Type)

Reservation (UserSSN, RoomCode, Date, StartTime, EndTime)

Room (RoomId, Type, CostPerHour)

1) Write a trigger that prevents the reservation of already booked rooms

We need to capture reservations for the same room with overlapping periods. We can use a“before” semantics to impose an early rollback of the transaction. (N.B.: two intervals are overlapped if the first begins before the end and ends before the beginning of the other)

```
create trigger ThouShallNotBook
before insert into Reservation
for each row
when exists ( select *
              from Reservation
              where RoomCode = new.RoomCode
                  and Date = new.Date and  StartTime < new.EndTime
                  and EndTime > new.StartTime )
rollback
```

Suppose that usage data are inserted into a table Usage only after the room has been **actually** used. Enrich the schema to track the number of hours that have been reserved but not used by each user, and write a (set of) trigger(s) that set the “type” of a user to “unreliable” when he totalizes 50 hours of unused reservations.

We track actual usage times and costs in the Usage table:

Usage (UserSSN, RoomCode, Date, StartTime, EndTime, Cost)

Unused hours can be counted via queries, without further schema modifications. For efficiency reasons, however, we may want to calculate the number incrementally, e.g., by adding a “WastedHours” field to table User.

How do we matche reservations and usages?

We assume that (1) the previous trigger guarantees reservations to be correct and not overlapping and that (2) actual usages of each room only and always happen within the limits of a reserved interval, and by those who actually reserved the room.

```
create trigger UpdateWastedHours
after insert into Usage
for each row
 update User
  set WastedHours = WastedHours +
                                  ( select EndTime – StartTime – ( new.EndTime – new.StartTime )
                                    from Reservation
                                    where RoomCode=new.RoomCode and Date = new.Date
                                        and StartTime>= new.StartTime and  EndTime <=
new.EndTime )
  where SSN = new.UserSSN
```

This is the simplest option: the field is updated at each insertion, possibly with a zero-increment when users are on time.
Capturing zero-increments may be done in the when clause (but is as heavy as the update)

The only missing part is the monitoring of the threshold:

```
create trigger UpdateType
after update of WastedHours on User
for each row
when old.WastedHours < 50 and new.WastedHours >= 50
do
    update User
    set Type = “Unreliable”
    where SSN = old.SSN
```

ATTENTION:
we’re not considering the case in which bands do not show up at all !

How to deal with users who don’t show up at all?

- We may not accept any new reservation for users who didn’t show up in the past (but this would be a new business rule – we should simply try to count the hours as wasted hours... What we miss, in this case, is the triggering event)
- We may consider a new reservation a triggering event and, before reserving, check for previous “dangling” reservations
    - And delete them, once dealt with (in order not to count them again in the future)
    - Or, more likely, in order not to delete potentially useful data, mark them with a flag that needs to be added to the schema
- Most likely (and simply), periodically check for these situations: we need a triggering event that is not a data modification, but rather a system event (example: check all reservations daily, after the change of date), as in the following trigger

```
create trigger InelegantNonPortable_GhostMusicians
after change-date()       // each vendor has its own extended event language
do
 select * into PENDING
 from Reservation R
 where R.date = today()–1 and not exists(  select * from Usage U
                                            where U.Date = R.Date
                                              and R.StartTime <= U.StartTime
                                              and U.EndTime >= R.EndTime )
 for-each X in PENDING
   do
      update User
      set WastedHours = WastedHours + X.EndTime – X.StartTime
end;
```

This solution pre-extracts the relevant reservations into a virtual table (PENDING) and uses a proprietary explicit iteration (for-each) in order to apply the modifications.
A much more elegant solution, using only SQL-2, is in the next trigger

```
create trigger ElegantAndPortable_GhostMusicians
after change-date()      // each vendor has its own extended event language
do
 update User U
  set WastedHours = WastedHours +
                                   ( select sum( P.EndTime – P.StartTime )
                                     from Reservation P
                                     where P.Date = today()–1 and P.UserSSN = U.SSN
                                         and not exists( select *
                                                                  from Usage S
                                                                  where S.Date = P.Date
                                                                            and P.StartTime <= S.StartTime
                                                                            and S.EndTime >= P.EndTime ) )
end;
```

Please note that the bindings on ***P*** *and* ***U*** *take the place of the iteration, and each user has its counter updated by the quantity depending on their specific "faults". Also note that this solution, by means of aggregation, also accounts for the case in which the same user has left more than one pending reservation in the same day.*

Esercizio

- Un sistema di regole attive calcola l'H-index dei ricercatori, definito come il massimo numero H di articoli scritti dal ricercatore che risultano citati da almeno H altri articoli; per cui, se un reicercatore ha pubblicato 5 articoli, rispettivamente citati da 7, 4, 4, 1 e 0 altri articoli, il suo H-index è uguale a 3. Si dispone di identificatori univoci Art-id (per gli articoli) e Ric-id (per i ricercatori), e due tabelle descrivono gli autori dei vari articoli (ogni articolo ha uno o più autori) e il numero delle citazioni:

AUTORE(Art-id, Ric-id)

CITAZIONI(Art-id, Cit-count)

- Gli unici eventi da considerare sono (a) l'incremento del campo Cit-count e (b) l'inserimento (transazionale) di un nuovo articolo, che si traduce in alcuni inserimenti nella tabella AUTORE e un inserimento in CITAZIONI. Si gestisca tramite regole attive l'aggiornamento dell'H-Index nella tabella

RICERCATORE(Ric-id, H-Index)

- Suggerimento: Si può ipotizzare che l'H-Index contenuto nella tabella RICERCATORE sia consistente con i dati delle altre tabelle all'inizio di ogni transazione, e ragionare in modo incrementale conservando la consistenza. L'inserimento/modifica di un solo articolo può infatti incrementare l'H-index di una sola unità.

Reazione all’evento (a)

L’incremento di citazioni di un articolo che aveva Cit-count già superiore all’H-index attuale dei suoi autori non può aumentarne l’H-index, così come non può aumentarlo un incremento che porta il nuovo Cit-count ad un nuovo valore ancora inferiore all’H-Index attuale. L’incremento dell’H-index si ha solo se old.Cit-count è inferiore (o uguale) all’H-Index e new.Cit-count è superiore, e ovviamente solo per (un sottoinsieme de-)gli autori dell’articolo che vede incrementate le sue citaizoni:

```
create rule Aggiorna_Update
after update of Cit-count on CITAZIONI
for each row
begin
  update RICERCATORE R
  set R.H-Index = R.H-Index + 1
  where old.Cit-count <= R.H-Index and new.Cit-count > R.H-Index and
                                                   // l’incremento fa superare la soglia
          ( R.Ric-id, new.Art-Id ) in ( select * from AUTORE )    // R è un autore dell’articolo in C
           and R.H-Index + 1 <= ( select count( * )
                                  from CITAZIONI C
                                  where ( R.Ric-id, C.Art-Id ) in ( select * from AUTORE )
                                         and C.Cit-count >= H-Index + 1 )
end
```

Reazione all’evento (b)

Con ipotesi sbrigativa (ma non irragionevole) si può sostenere che l’inserimento dei nuovi articoli avviene sempre a breve distanza dalla loro prima pubblicazione, quindi il lor Cit-count è necessariamente inizializzato a zero, e quindi questii inserimenti non possono incrementare l’H-Index (saranno i successivi incrementi del Cit-count a far scattare il trigger definito per (a))
Se invece si ammete che la base di dati possa essere estesa anche inserendo articoli molto tempo dopo la loro data di pubblicazione, allora occorre ricalcolare l’H-Index per tutti gli autori dell’articolo incrementato:

```
create rule Aggiorna_Insert
after insert into CITAZIONI
for each row
do
  update RICERCATORE R
  set R.H-Index = R.H-Index + 1
  where R.H-Index + 1 = ( select count( * )
                          from CITAZIONI C
                          where ( R.Ric-id, C.Art-Id ) in ( select * from AUTORE )
                             and C.Cit-count >= H-Index + 1 )
  and ( R.Ric-id, new.Art-Id ) in ( select * from AUTORE )
end
```

FlyWithMe

Consider a database for an airline:

AIRPORT (Name, Country, City)

FLIGHT (Code, DepAirport, DepDate, DepTime, ArrAirport, ArrDate, ArrTime)

Assume the database is distributed in the various cities in which the company operates.

1) Describe a reasonable data fragmentation that allows each node to locally execute the query that extracts the destinations directly reachable from an airport.

2) Write at the fragmentation and language transparency levels the query that extracts for every nation the average number of flights that depart from cities of that nation.

3) Write at the fragmentation and language transparency levels the update that, due to bad weather, moves all the arrivals of the day 04/04/2014 from Madrid to Barcelona.