COMP9311 Week 07 Lecture

Triggers

Triggers (review)

Triggers are actions invoked by DB modifications.

They allow programmers to

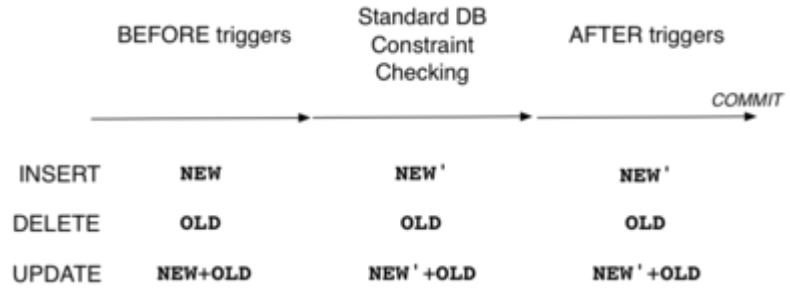
- implement global constraint (assertion) checking
- maintain summary values (cross-table dependencies)

They achieve this by

- invoking functions before/after insert/delete/update
- using/manipulating `OLD`/`NEW` values of changed tuples

... Triggers (review)

Sequence of activities during database update:



Note: `BEFORE` trigger can modify value of new tuple

Triggers in PostgreSQL

PostgreSQL triggers provide a mechanism for

- `INSERT`, `DELETE` or `UPDATE` events
- to automatically activate PLpgSQL functions

Syntax for PostgreSQL trigger definition:

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [OR Event2 ...]
ON TableName
[ WHEN ( Condition ) ]
FOR EACH {ROW|STATEMENT}
EXECUTE PROCEDURE FunctionName(args...);
```

... Triggers in PostgreSQL

There is no restriction on what code can go in the function.

However a `BEFORE` function must contain one of:

```
RETURN old;    or    RETURN new;
```

depending on which version of the tuple is to be used.

If `BEFORE` trigger returns `old`, no change occurs.

If exception is raised in trigger function, no change occurs.

Trigger Example #1

Consider a database of people in the USA:

```
create table Person (
   id      integer primary key,
   ssn     varchar(11) unique,
   ... e.g. family, given, street, town ...
   state   char(2), ...
);
create table States (
   id      integer primary key,
   code    char(2) unique,
   ... e.g. name, area, population, flag ...
);
```

Constraint: `Person.state ∈ (select code from States)`, or
`exists (select id from States where code=Person.state)`

... Trigger Example #1

Example: ensure that only valid state codes are used:

```
create trigger checkState before insert or update
on Person for each row execute procedure checkState();

create function checkState() returns trigger as $$
begin
   -- normalise the user-supplied value
   new.state = upper(trim(new.state));
   if (new.state !~ '^[A-Z][A-Z]$') then
      raise exception 'Code must be two alpha chars';
   end if;
   -- implement referential integrity check
   select * from States where code=new.state;
   if (not found) then
      raise exception 'Invalid code %',new.state;
   end if;
   return new;
end;
$$ language plpgsql;
```

... Trigger Example #1

Examples of how this trigger would behave:

```
insert into Person
   values('John',...,'Calif.',...);
-- fails with 'Statecode must be two alpha chars'

insert into Person
   values('Jane',...,'NY',...);
-- insert succeeds; Jane lives in New York

update Person
   set town='Sunnyvale',state='CA'
         where name='Dave';
-- update succeeds; Dave moves to California

update Person
   set state='OZ' where name='Pete';
-- fails with 'Invalid state code OZ'
```

Trigger Example #2

Example: department salary totals

Scenario:

```
Employee(id, name, address, dept, salary, ...)
Department(id, name, manager, totSal, ...)
```

An assertion that we wish to maintain:

```
create assertion TotalSalary check (
   not exists (
       select d.id from Department d
       where  d.totSal <>
              (select sum(e.salary)
               from Employee e
               where e.dept = d.id)
    )
)
```

... Trigger Example #2

Events that might affect the validity of the database

- a new employee starts work in some department
- an employee gets a rise in salary
- an employee changes from one department to another
- an employee leaves the company

A single assertion could check for this after each change.

With triggers, we have to program each case separately.

Each program implements updates to *ensure* assertion holds.

... Trigger Example #2

Implement the Employee update triggers from above in PostgreSQL:

Case 1: new employees arrive

```
create trigger TotalSalary1
after insert on Employees
for each row execute procedure totalSalary1();

create function totalSalary1() returns trigger
as $$
begin
    if (new.dept is not null) then
        update Department
        set    totSal = totSal + new.salary
        where  Department.id = new.dept;
    end if;
    return new;
end;
$$ language plpgsql;
```

... Trigger Example #2

Case 2: employees change departments/salaries

```
create trigger TotalSalary2
after update on Employee
for each row execute procedure totalSalary2();

create function totalSalary2() returns trigger
as $$
begin
```

```
    update Department
    set    totSal = totSal + new.salary
    where  Department.id = new.dept;
    update Department
    set    totSal = totSal - old.salary
    where  Department.id = old.dept;
    return new;
end;
$$ language plpgsql;
```

... Trigger Example #2

Case 3: employees leave

```
create trigger TotalSalary3
after delete on Employee
for each row execute procedure totalSalary3();

create function totalSalary3() returns trigger
as $$
begin
    if (old.dept is not null) then
        update Department
        set    totSal = totSal - old.salary
        where  Department.id = old.dept;
    end if;
    return old;
end;
$$ language plpgsql;
```

Exercise: Triggers (1)

Requirement: maintain assets in bank branches

- each branch has assets based on the accounts held there
- whenever an account changes, the assets of the corresponding branch should be updated to reflect this change

Some possible changes:

- a new account is opened
- the amount of money in an account changes
- an account moves from one branch to another
- an account is closed

Implement triggers to maintain `Branch.assets`

[Solutions]

Exercise: Triggers (2)

Consider a simple airline flights/bookings database:

```
Airports(id, code, name, city)
Planes(id, craft, nseats)
Flights(id, fltNum, plane, source, dest
        departs, arrives, price, seatsAvail)
Passengers(id, name, address, phone)
Bookings(pax, flight, paid)
```

Write triggers to ensure that `Flights.seatsAvail` is consistent with number of `Bookings` on that flight.

Assume that we never `UPDATE` a booking (only insert/delete)

[Solutions]

Programming with DBs

Programming with Databases

So far, we have seen ...

- accessing data via SQL queries
- packaging SQL queries as views/functions
- building functions to return tables
- implementing assertions via triggers

All of the above programming

- is very close to the data
- takes place inside the DBMS

... Programming with Databases

Complete applications require code outside the DBMS

- to handle the user interface (GUI or Web)
- to interact with other systems (e.g. other DBs)
- to perform compute-intensive work (vs. data-intensive)

"Conventional" programming languages (PLs) provide these.

... Programming with Databases

Requirements of an interface between PL and RDBMS:

- mechanism for connecting to the DBMS
- mapping betwen tuples and PL objects
- mechanism for mapping PL "requests" to queries
- mechanism for iterating over query results

Distance between PL and DBMS is variable, e.g.

- `libpq` allows C programs to use PG structs
- JDBC transmits SQL strings, retrieves tuples-as-objects

PL/DB Interface

Common DB access API used in programming languages

```
db = connect_to_dbms(DBname,User/Password);
query = build_SQL("SqlStatementTemplate",values);
results = execute_query(db,query);
while (more_tuples_in(results))
{
   tuple = fetch_row_from(results);
   // do something with values in tuple ...
}
```

This pattern is used in many different libraries:

- Java/JDBC, PHP/PDO, Perl/DBI, Python/dbapi2, Tcl, ...

... PL/DB Interface

DB access libraries have similar overall structure.

However, they differ in the details:

- whether specific to one database or generic

- whether object-oriented or procedural flavour
- function/method names and parameters
- how to get data from program into SQL statements
- how to get data from tuples to program variables

We use PHP to illustrate the idea in this lecture.

PHP/DB Interface

Standard pattern for extracting data from DB:

```
$db = dbConnect("dbname=myDB");
...
$query = "select a,b,c from R where c >= %d";
$result = dbQuery($db, mkSQL($query, $min));
while ($tuple = dbNext($result)) {
   $tmp = $tuple["a"] - $tuple["b"] - $tuple["c"];
   # or ...
   list($a,$b,$c) = $tuple;
   $tmp = $a - $b - $c;
}
...
```

DB Library

Functions in the database library:

- dbConnect(conn): establish connection to DB
- dbQuery(db,sql): send SQL statement for execution
- dbNext(res): fetch next tuple from result set
- dbUpdate(db,sql): send SQL insert/delete/update
- ...

Most functions terminate with message if error occurs.

... DB Library


```
$t = dbNext(resource $r);
```

- `$t` is assigned next tuple from result set `$r`
- `$t` contains two copies of values from tuple
  - one set of values is indexed by position in SELECT clause
  - one set of values is indexed by name in SELECT clause

Example:

```
$q = "select name,max(mark) from Enrolments ...";
$r = dbQuery($db,$q);
$t = dbNext($r);
# results in $t with value
array(0=>'John', "name"=>'John', 1=>95, "max"=>95)
```

Example PHP code (actual code)


```
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if ($result) {
  echo "The query executed successfully.\n";
  for ($row = 0; $row < pg_numrows($result); $row++) {
    $fullname = pg_result($result, $row, 'title') . " ";
    $fullname .= pg_result($result, $row, 'fname') . " ";
    $fullname .= pg_result($result, $row, 'lname');
    echo "Customer: $fullname\n";
```

```
  }
} else {
  echo "The query failed with the following error:\n";
  echo pg_errormessage($db_handle);
}
pg_close($db_handle);
```

DB/PL Mismatch

There is a tension between PLs and DBMSs

- DBMSs deal efficiently with sets of tuples
- PLs encourage dealing with single tuples/objects

If not handled carefully, can lead to inefficient use of DB.

Note: relative costs of DB access operations:

- establishing a DBMS connection ... very high
- initiating an SQL query ... high
- accessing individual tuple ... low

... DB/PL Mismatch

Consider the PL/DBMS access method, phrased in PHP:

```
--  establish connection to DBMS
$db = dbAccess("DB");
$query = "select a,b from R,S where ... ";
--  invoke query and get handle to result set
$results = dbQuery($db, $query);
--  for each tuple in result set
while ($tuple = dbNext($results)) {
    --  process next tuple
    process($tuple['a'], $tuple['b']);
}
```

... DB/PL Mismatch

Example: find mature-age students

```
$query = "select * from Student";
$results = dbQuery($db,$query);
while ($tuple = dbNext($results)) {
    if ($tuple["age"] >= 40) {
        --  process mature-age student
    }
}
```

If 10000 students, and only 500 of them are over 40, we transfer 9500 unnecessary tuples from DB.

... DB/PL Mismatch

E.g. should be implemented as:

```
$query = "select * from Student where age >= 40";
$results = dbQuery($db,$query);
while ($tuple = dbNext($results)) {
    --  process mature-age student
}
```

Transfers only the 500 tuples that are needed.

... DB/PL Mismatch

Example: find info about all marks for all students

```
$query1 = "select id,name from Student";
$res1 = dbQuery($db,$query1);
while ($tuple1 = dbNext($res1)) {
    $query2 = "select course,mark from Marks".
              " where student = $tuple1['id']";
    $res2 = dbQuery($db,$query2);
    while ($tuple2 = dbNext($res2)) {
        --  process student/course/mark info
    }
}
```

If 10000 students, we invoke 10001 queries on the database.

... DB/PL Mismatch

E.g. should be implemented as:

```
$query = "select id,name,course,mark".
         " from Student s, Marks m".
         " where s.id = m.student";
$results = dbQuery($db,$query);
while ($tuple = dbNext($results)) {
    --  process student/course/mark info
}
```

We invoke 1 query, and transfer same number of tuples.

Produced: 27 August 2017