Aggregate Functions & Nested Queries

Dr. Munesh Singh

Indian Institute of Information Technology
Design and Manufacturing,
Kancheepuram
Chennai-600127

February 15, 2019

Agrregate functions

- These functions operate on the multiset of values of a column of a relation, and return a value
  1. avg: average value
  2. min: minimum value
  3. max: maximum value
  4. sum: sum of values
  5. count: number of values

Agrregate functions

- Find the average salary of instructors in the computer science department
  select avg(slaray) from instructor where deptname='cse';
- Find the total number of the instructor who teach a course in Spring 2018 semester
  select count(distinct ID) from teaches where semester='Spring' and year='2010';
- Find the number of tuple in the course relation
  select count(*) from course;

Agrregate functions-Group By and having

- Find the salary of instructors in each department
  select deptname, avg(slaray) as avgs from instructor group by deptname;
- Find the names and average salaries of all departments whose salary is greater than 42000
  select deptname, avg(salary) from instructor group by deptname having avg(salary)>42000;
- **Note:** predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups

Null Values and Aggregates

- Total all salaries
  select sum(slaray) from instructor;
- Above statement ignores null amounts
- Result in null if there is no non-null amount
- All aggregate operations except count(*) ignore tuples with null values on the aggregated attributes
- what if collection has only null values?
  - count return 0
  - all other aggregates return null

Nested Sub-Queries

- SQL provides a mechanism for the nesting of subqueries
- A subquery is a **select-from-where** expression that is nested within another query
- The nesting can be done in the following SQL query
  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_n$
  where P
- as follows:
- A_i can be replaced be a subquery that generate a single value
- r_i can be replaced by any valid subquery
- P can be replaced with an expression of the form:
  - B< *operation* >(subquery)
  - where B is an attribute and < *operation* > to be defined later

Sub-Queries in the Where Clause

- A common use of subqueries is to perform tests:
  - For set membership
  - For set comparisons
  - For set cardinality

Set Membership

- Find courses offered in Fall 2009 and in Spring 2010
  select distinct courseid from section where semester='Fall' and year=2009 and courseid in (select courseid from section where semester='Spring' and year=2010)
- Find courses offered in Fall 2009 and not in Spring 2010
  select distinct courseid from section where semester='Fall' and year=2009 and courseid not in (select courseid from section where semester='Spring' and year=2010)

Set Membership Cont...

- Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101
  select count(distinct ID) from takes(courseid,secid,semester,year) in (select courseid secid,semester,year from teaches where teaches.ID=10101)

Set Comparison -"some" Clause

- Find the names of instructor with salary greater than that of some (at least one) instructor in the Biology department
  select distinct T.name from instructor as T, instructor as S where T.salary¿S.salary and S.deptname='Biology';
- Same query using $>$ some clause
  select name from instructor where salary¿ some (select salary from instructor where dept name='Biology');

Set Comparison -"all" Clause

- Find the names of instructor whose salary is greater than the salary of all instructor in Biology department
  select name from instructor where salary> all (select salary from instructor where dept name='Biology');

Test for Empty Relations

- There **exists** construct return the value true if the argument subquery is nonempty
- exist r $\Leftrightarrow$ r $\neq \Phi$
- not exist r $\Leftrightarrow$ r$=\Phi$ '
- Find courses taught in both Fall 2009 and in Spring 2010 semester

  select courseid from section as S where semester='Fall' and year=2009 and exists (select * from section as T where semester='Spring' and year=2010 and S.courseid=T.courseid);

Use of "not exists" Clause

- Find all students who have taken all courses offered in the Biology department.
  select distinct S.ID,S.name from student as S where not exists((select courseid from course where deptname='Biology') except (select T.courseid from takes as T where S.ID=T.ID))
- First nested query list all courses offered in Biology
- Second nested query lists all courses a particular student took

Test for Absence of Duplicate Tuples

- The unique construct tests whether a subquery has any duplicates tuple in its result
- The unique construct evalute to "true" if a given subquery constains no duplicates
- Find all courses that were offered at most once in 2009
  select T.courceid from course as T where unique(select R.courseif from section as R where T.courseid=R.courseid and R.year=2009);

Subquieries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructor salaries of those departments where the average salary is greater than 42000.
  select deptname,avgs from (select deptname, avg(salary) as avgs from instructor group by deptname) where avgs>42000;
- same query in another way
  select deptname,avgs from (select deptname, avg(salary) as from instructor group by deptname) as deptavg(dname,avgs) where avgs>42000;

With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs
- Find all department with the maximum budget
  with maxbudget(value) as(select max(budget) from department) select department.name from department, maxbudget where department.budget=maxbudget.value

Complex Queries using With Clause

- Find all department where the total salary is greater than the average of the total salary at all departments
  with depttotal(deptname,value) as (select deptname, sum(salary) from instructor group by deptname), depttavg(value) as(select avg(value) from depttotal) select deptname from depttotal,depttavg where depttotal.value> depttavg.value;

Subquaries in the Select Caluse

Scalary Subquery

- Scalar subquery is one which is used where a single value is expected
- List all department along with the number of instructors in each department

select deptname,(select count(*) from instructor where department.deptname=instructor.deptname) as numinstructors from department

Modification of the Database

- Deletion of tuples from a given relation
- Inserting of new tuples into a given relation
- Updating of values in some tuples in a given relation
- delete all instructors
  delete from instructor
- delete all instructor from Finance department
  delete from instructor where deptname='Finance'

Modification of the Database

- Delete all tuples on the instructor relation for those instructors associated with a department located in the Watson building

  delete from instructor where deptname in (select deptname from department where building='Watson');

Modification of the Database

- Delete all instructors whose salary is less than the average salary of instructors
  delete from instructor where salary<(select avg(salary) from instructor);
- **Problem**: as we delete tuples from deposit, the average salary changes
- **Solution used in SQL**
  - First, compute avg(salary) and find all tuples to delete
  - Next, delete all tuples found above (without recomputing avg or retesting the tuples)

Insertion

- Add a new tuple to course
  insert into course values('cs-117','Database System', 'CSE',4);
- or equivalently
  insert into course (courseid,title,deptname,credit) values('cs-117','Database System', 'CSE',4);
- Add a new tuple to student with totalcredit set to null
  insert into student values('3003','Green', 'Finance',0);

Insertion (Cont...)

- Add all instructors to the student relation with totalcredit set to 0
  insert into student select ID, name,deptname,0 from instructor
- The select from where statement is evaluated fully before any of its results are inserted into the relation

Updates

- Increase salaries of instructors whose salary is over 1000000 by 3% and all others by a 5%
- Write two update statements:
  update instructor set salary=salary*1.03 where salary> 100000;
  update instructor set salary=salary*1.05 where salary<= 100000;
- The order is important
- Can be done better using the case statement

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor set salary=case
when salary<= 100000 then salary*1.05
else salary*1.03
end
```

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all actual stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation describe, in SQL by
  select ID, name, deptname from instructor
- A view provides a mechanism to hide certain data from the view of certain users
- relation that is not of the conceptual model but is made available to a user as a "virtual relation" is called a view

Views Definition

- A view is defined using the create view statement which has the form
  create view v as $<$ *query expression* $>$
- where $<$ *queryexpression* $>$ is any legal SQL expression
- The view name is represented by v
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view

Example Views

- A view of instructors without their salary **create view faculty as select id,name,deptname from instructor**
- Find all instructors in Biology department **select name from faculty where deptname='Biology'**
- Create a view of department salary totals **create view departmenttotalsalary(deptname, totalsalary) as select deptname, sum(salary) from instructor group by deptname**

Views Defined Using Other Views

- **create view physics_fall_2009_watson as select courseid,roomno fromselect course.courseid,secid,building,roomno from (select course.courseid,secid,building,roomno course,section where course.courseid=section.courseid and course.deptname='Physics' and section.semester='Fall' and section.year='2009');**

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation V_1 is said to **depend directly** on a view relation V_2 if V_2 is used in the expression defining V_1
- A view relation V_1 is said to **depend on** view relation V_2 if either V_1 depends directly to V_2 or there is a path of dependencies from V_1 to V_2
- A view relation v is said to be recursive if it depends on itself

Views Expansion

- A way to define the meaning of views defined in terms of other views
- Let view V_1 be defined by an expression e_1 that may itself contain uses of view relations
- View expansion of an expression repeats the following replacement step:
  repeat
  Find any view V_i in e_1
  Replace the view relation V_j by the expression defining V_i
  until no more view relations are present in e_1
- As long as the view definitions are not recursive, this loop will terminate

Recursive View

- In SQL, recursive queries are typically built using these components:
  1. A non-recursive seed statement
  2. A recursive statement
  3. A connection operator
     The only valid set connection operator in a recursive view definition is UNION ALL
  4. A terminal condition to prevent infinite recursion
- View expansion of an expression repeats the following replacement step:
  repeat
  Find any view V_i in e_1
  Replace the view relation V_j by the expression defining V_i
  until no more view relations are present in e_1
- As long as the view definitions are not recursive, this loop will terminate

Recursive View-Example

- In the context of a relation flights:
- Find all the destinations that can be reached from 'Paris'

```
create table flights (
    source      varchar(40),
    destination varchar(40),
    carrier     varchar(40),
    cost        decimal(5,0));
```

source	destination	carrier	cost
Paris	Detroit	KLM	7
Paris	New York	KLM	6
Paris	Boston	American Airlines	8
New York	Chicago	American Airlines	2
Boston	Chicago	American Airlines	6
Detroit	San Jose	American Airlines	4
Chicago	San Jose	American Airlines	2

Recursive View-Example

- **create recursive view reachable_from(source,destination,depth) as (select root.source,root.destination, 0 as depth from flight as root where root.source='Paris' union all select in1.source, out1.destination,in1.depth+1 from reachable_from as in1,flights as out1 where in1.destination=out1.source and in1.depth<= 100);**

Recursive View-Example

Result of the previous query

- Get the result by simple selection on the view:
  select distinct source,destination from reachable_from

source	destination
Paris	Detroit
Paris	New York
Paris	Boston
Paris	Chicago
Paris	San Jose

source	destination	carrier	cost
Paris	Detroit	Jet	7
Paris	New York	Jet	6
Paris	Boston	indigo	8
New York	Chicago	indigo	2
Boston	Chicago	indigo	6
Detroit	San Jose	indigo	4
Chicago	Detroit	indigo	7

The Power of Recursion

- Recursive View makes it possible to write queries, such as transitive closure queries, that can not be written without recursion or iteration
- Intution: Without recursion, a non-recursive non interative program can perform only a fixed number of joins of flights with itself
  - This can give only a fixed number of levels of reachable destinations
  - Given a fixed non-recursive query, we can construct a database with a greater number of levels of reachable destinations on which the query will not work

The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to reachable_from
  - The next slide shows a flights relation
  - Each step of the iterative process constructs an extended version of reachable_from its recursive definition
  - The final result is called the fixed point of the recursive view definition.
- Recursive view are required to be monotonic. That is, if we add tuples to flights the view reachable_from contains all of the tuples it contained before,plus possible more

The Power of Recursion

source	destination	carrier	cost
Paris	Detroit	Jet	7
Paris	New York	Jet	6
Paris	Boston	indigo	8
New York	Chicago	indigo	2
Boston	Chicago	indigo	6
Detroit	San Jose	indigo	4
Chicago	Detroit	indigo	7

Iteration#	Tuples in closure
0	Detroit,New York,Boston
1	Detroit, New York, Boston, San Jose, Chicago
2	Detroit, New York, Boston, San Jose, Chicago

Update of a View

- Add a new tuple to faculty view which we defined earlier
  insert into faculty values('30765','Green','Music');
  This insertion must be represented by the insertion of the tuple ('30765','Green','Music',null)
  into the instructor relation

Some Updates cannot be Translated Uniquely

- **create view instructor_info as**
  select ID, name, building from instructor, department
  where instructor.dept_name=department.dept_name;
- **insert into instructor_info**
  values('69987','White','Taylor');
  - which department, if multiple department in Taylor?
  - what if no department is in Taylor?
- Most SQL implementations allow updates only on simple views
  - The **form** clause has only one database relation
  - The **select** caluse contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause

Authorization

- Forms of authorization on parts of the database:
  - **Read**- allows reading, but not modification of data
  - **Insert**- allows insertion of new data, but not modification of exisitng data
  - **Update**- allows modification, but not deletion of data
  - **Delete**- allows deletion of data
- Forms of authorization to modify the database schema
  - **Index**- allows creation and deletion of indices
  - **Resources**- allows creation of new relations
  - **Alteration**- allows addition or deletion of attributes in a relations
  - **Drop**- allows deletion of relations

Authorization

- The grant statement is used to confer authorization:
  - **grant** $<$ *privilegelist* $>$
    on $<$ *relationnameorviewname* $>$ **to** $<$ *userlist* $>$
- $<$ *userlist* $>$ is:
  - **grant** a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator)

Some Updates cannot be Translated Uniquely

- **select:** allows read access to relation, or the ability to query using the view
- Example: grant users U_1, U_2, U_3 select authorization on the instructor relation:
  grant select on instructor to U_1, U_2, U_3
- **insert:** the ability to insert tuples
- **update:** the ability to update using the SQL update statement
- **delete:** the ability to delete tuples.
- **all privileges:** used as a short from for all the allowable privileges

Revoking Authorization in SQL

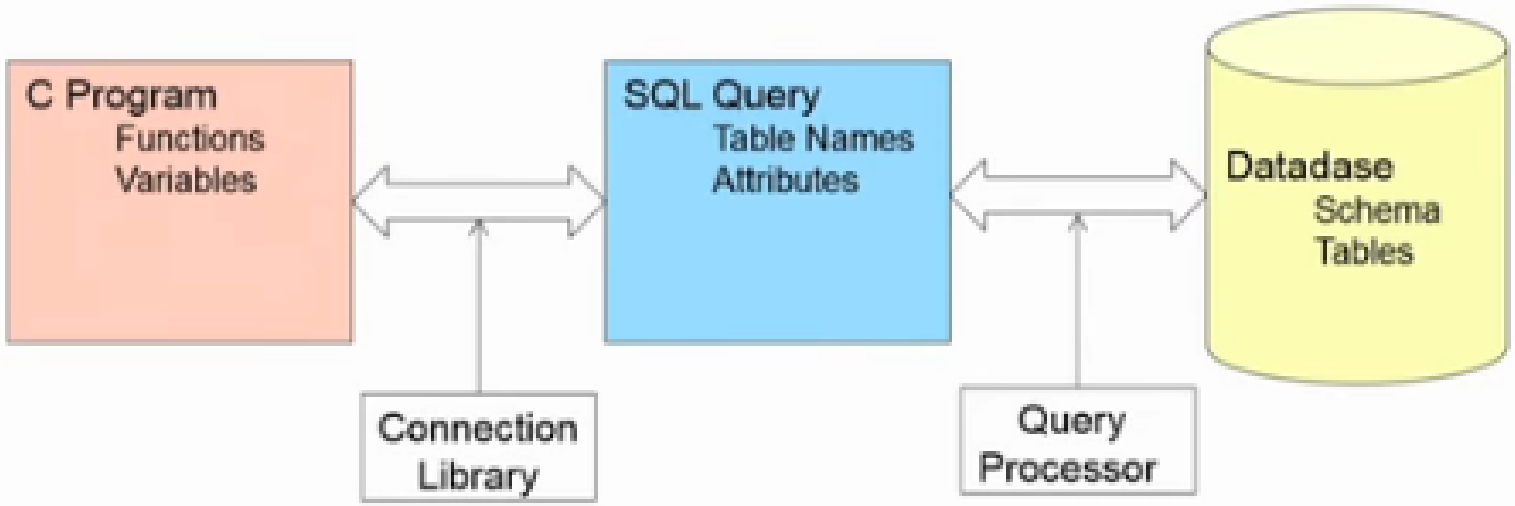
- The **revoke** statement is used to revoke authorization
  revoke$<$ *privilegelist* $>$
  on $<$ *relationnameorviewname* $>$ **from** $<$ *userlist* $>$
- Example:
  revoke select on branch from U_1, U_2, U_3
- $<$ *privilege* $-$ *list* $>$ may be all to revoke all privileges the revokee may hold
- if $<$ *revokee* $-$ *list* $>$ includes **public**, all users lose the privilege except those granted it explicitly
- if the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation
- All privileges that depends on the privilege are being revoke are also revoked

Roles

- **create role** instructor;
  grant instructor to Amit;
- Privileges can be granted to roles:
  grant select on takes **to** instructor;
- Roles can be granted to users, as well as to other roles
  create role teaching_assistant
  grant teaching_assistant to instructor;
- Instructor inherits all privileges to teaching_assistant
- Chain of roles
  - **create role** dean;
  - **grant** instructor **to** dean;
  - **grant** dean to ram;

Accessing SQL From A Programming Language

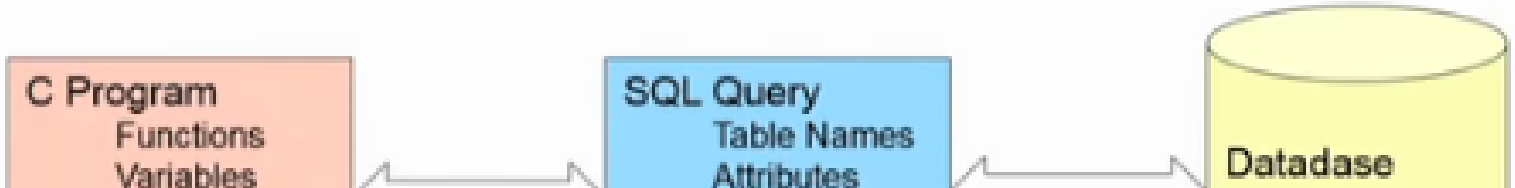
Native Language $<--->$ Query Language



Accessing SQL From A Programming Language

Native Language $<--->$ Query Language

- API (application programming interface) for a program to interact with a database server
- Application makes calls to
  - connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- Various Tools:
  - JDBC (Java Database Connectivity) work with java
  - ODBC (open Database Connectivity) works with C, C++, C#, Python.
  - Embedded SQL



JDBC

Native Language < −− > Query Language

- JDBC is a java API for communciating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the statement object to send queries and fetch results
  - Exception mechanism to handle error

ODBC

Native Language < −− > Query Language

- Open Database Connectivity (ODBC) standard
  - standard for application program to communicate with a database server
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results

Application Program Connection With ODBC

Native Language < −− > Query Language

Application Program

(MySQLdb is a thin Python wrapper around _mysql.
It is compatible with the Python DB API)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb
import sys
try:
    con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')
    cur = con.cursor()
    cur.execute("SELECT VERSION()")
    ver = cur.fetchone()
    print "Database version : %s " % ver

except mdb.Error, e:
    print "Error %d: %s" % (e.args[0],e.args[1])
    sys.exit(1)
finally:
        if con:
        con.close()
```

Database Query

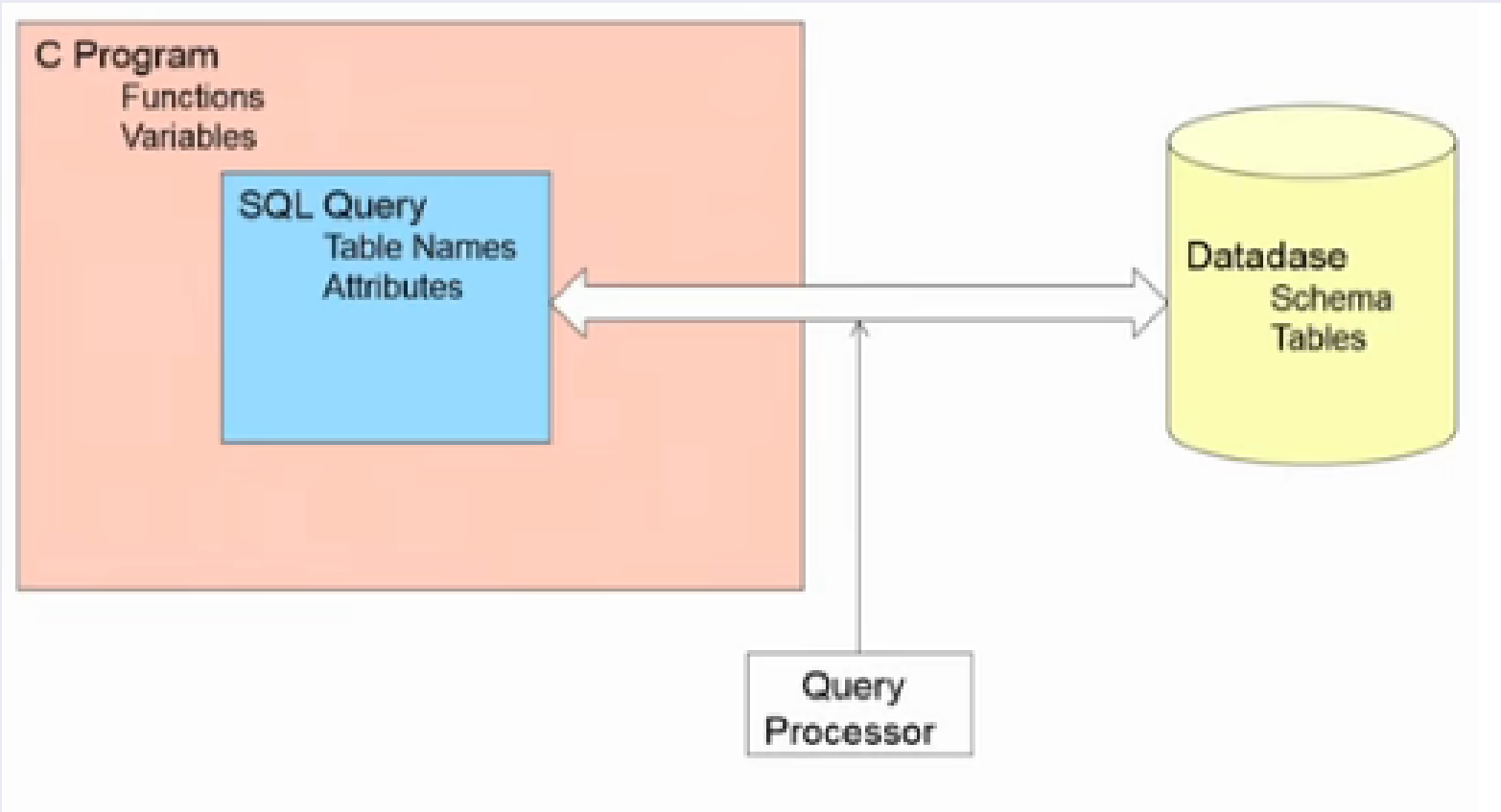
```
mysql> CREATE DATABASE testdb;
mysql> CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'test623';
mysql> USE testdb;
mysql> GRANT ALL ON testdb.* TO 'testuser'@'localhost';
mysql> quit;
```

```
cur = con.cursor()
cur.execute("DROP TABLE IF EXISTS Writers")
cur.execute("CREATE TABLE Writers(Name VARCHAR(25))")
cur.execute("INSERT INTO Writers(Name) VALUES('Jack London')")
cur.execute("INSERT INTO Writers(Name) VALUES('Honore de Balzac')")
```

SQL Part of Application Programming Language(Embedded SQL)

Dr. Munesh Singh

Native Language < −− > Query Language



Triggers)

Native Language $<--->$ Query Language

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database
- To design a trigger mechansim, we must:
  - specify the condition under the trigger is to be executed.
  - specify the actions to be taken when trigger executes
- Trigger introduced to SQL standard in:1999, but support even earlier using non-standard syntax by most databases

Triggering Events and Actions in SQL)

Native Language $<--->$ Query Language

- Triggering events can be **insert, delete, updates** item Triggers on update can be restricted to specific attributes
  - For example, **after update of** takes on grade
- Values of attributes before and after an update can be referenced
  - **referencing old row as:** for deletes and updates
  - **referencing new row as:** for inserts and updates
- Trigger can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

Trigger Example

- **create trigger setnull_trigger before update of takes**
  referencing new row as nrow
  for each row
  where (nrow.grade=”)
  begin atomic
  set nrow.grade=null
  end;

When NoT To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g. total salary of each department)
  - Replicating databases by recording changes to special relations (called change or delta relations) and having a seperate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintian summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

When NoT To Use Triggers

- Risk of unintended execution of trigger, for example , when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions