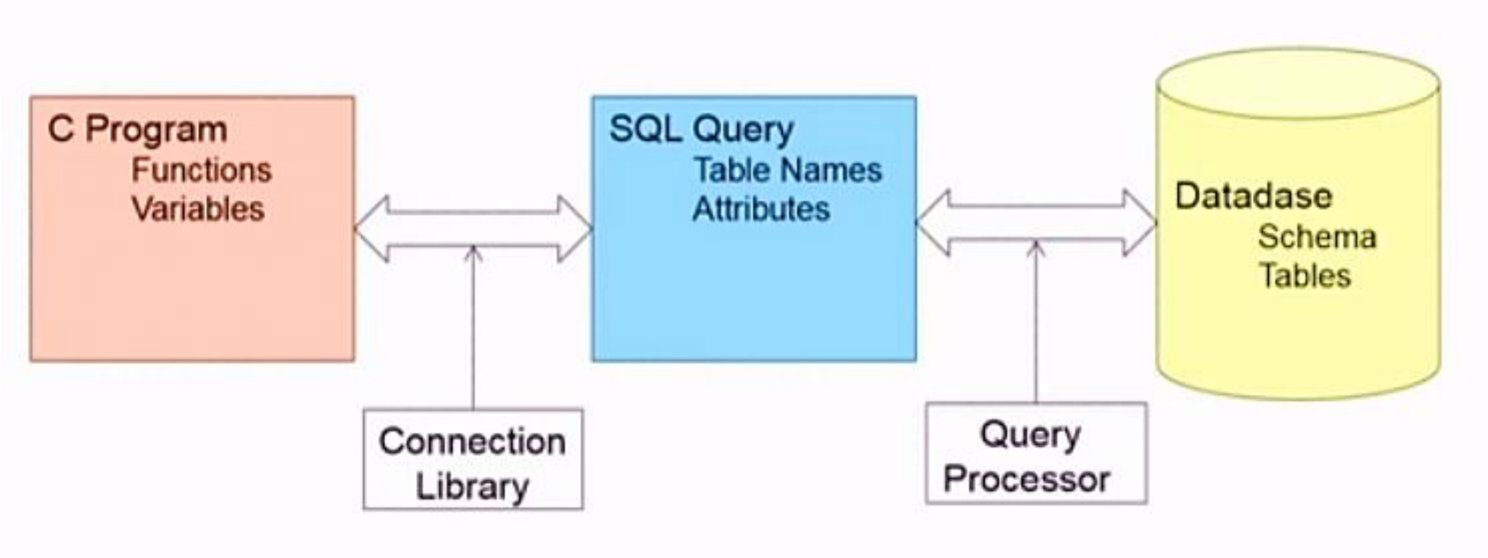
Advanced SQL

Native Language ↔ Query Language



Accessing SQL From a Programming Language

- API (application-program 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) works with Java
  - ODBC (Open Database Connectivity) works with C, C++, C#, Visual Basic, and Python
    - Other API’s such as ADO.NET sit on top of ODBC
  - Embedded SQL

JDBC

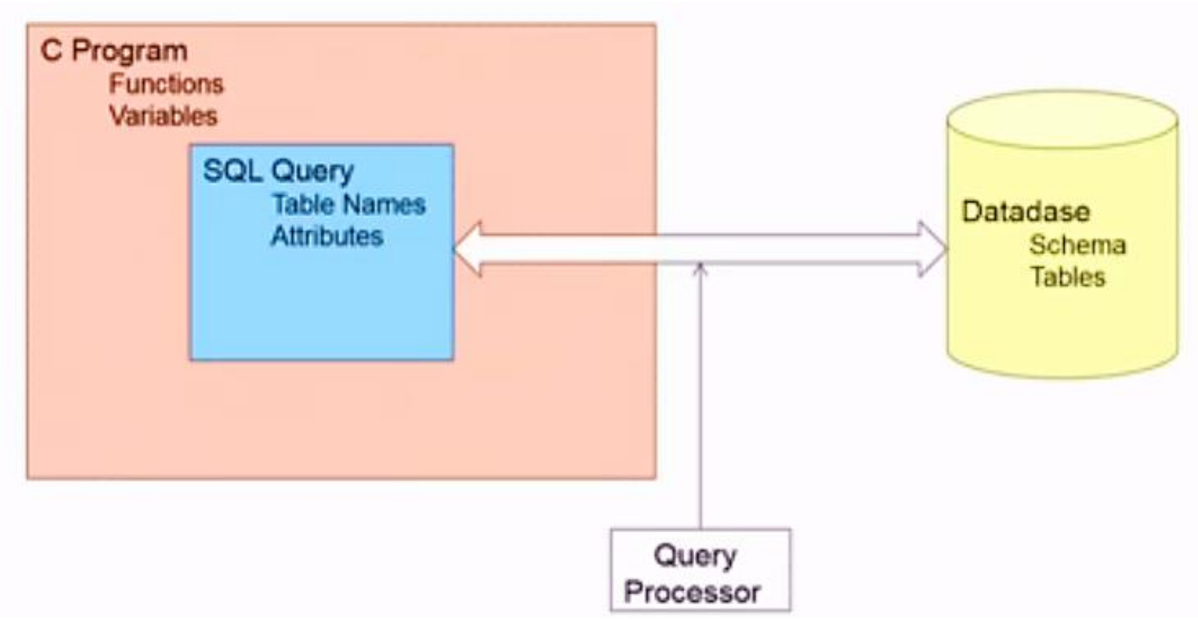
- **JDBC** is a Java API for communicating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types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

ODBC

- 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

- Applications such as GUI, spreadsheets, etc. can use ODBC

Native Language ↔ Query Language



Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL
- The basic form of these languages follows that of the System R embedding of SQL into PL/1
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement>;

  Note: This varies by language:

  – In some languages, like COBOL, the semicolon is replaced with END-EXEC

  – In Java embedding uses # SQL { …. };

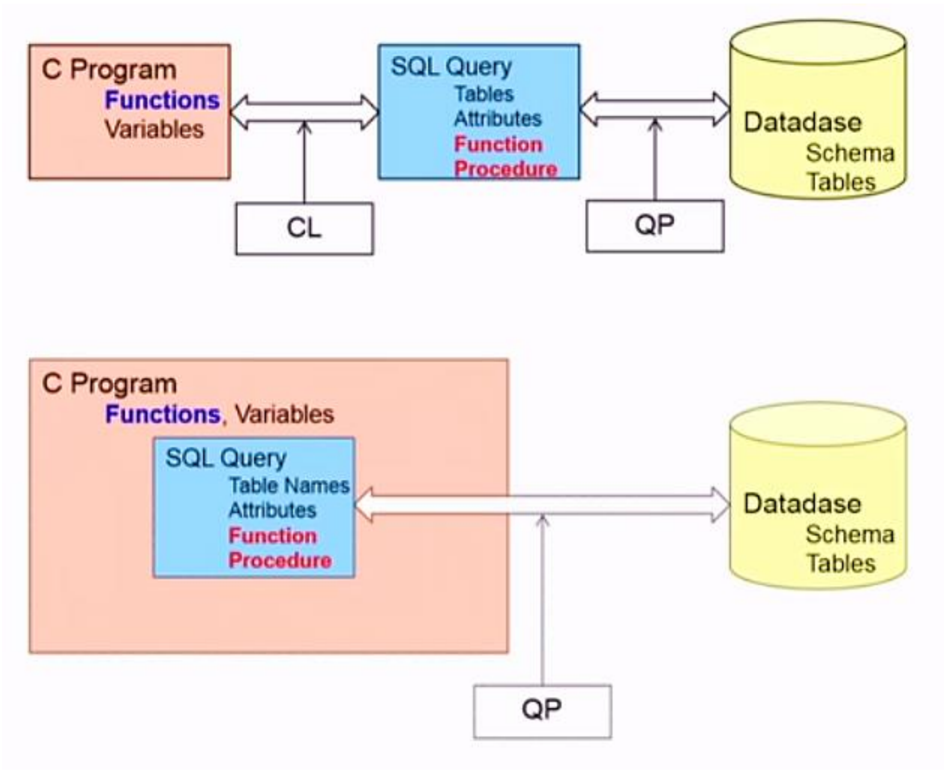
- Before executing any SQL statements, the program must first connect to the database
- This is done using:

  EXEC-SQL **connect to** *server* **user** *user-name* **using** *password*;

- Here, *server* identifies the server to which a connection is to be established

Functions And Procedural Constructs

- Native Language ↔ Query Language



SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department

```
create function dept_count (dept_name varchar(20))
   returns integer
  begin
      declare d_count  integer;
             select count (* ) into d_count
             from instructor
             where instructor.dept_name = dept_name
      return d_count;
  end
```

- The function *dept*_count can be used to find the department names and budget of all departments with more than 12 instructors

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

Triggers

- 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 mechanism, we must:
  - Specify the conditions under which the trigger is to be executed
  - Specify the actions to be taken when the trigger executes

- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases

- Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**

- 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

- Triggers can be activated before an event, which can serve as extra constraints
  - For example, convert blank grades to null

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
          set nrow.grade = null;
end;
```

Trigger to Maintain *credits_earned* Value

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
        and (orow.grade = 'F' or orow.grade is null)
begin atomic
        update student
        set tot_cred = tot_cred +
                (select credits
                 from course
                where course.course_id= nrow.course_id)
        where student.id = nrow.id;
end;
```

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

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 separate 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 maintain 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 triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site

  Trigger execution can be disabled before such actions

- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

Next Lecture

Normalization

Thank you for your attention...

Any question?

Contact:
Department of Information Technology, NITK Surathkal, India
6th Floor, Room: 13
Phone: +91-9477678768
E-mail: shrutilipi@nitk.edu.in