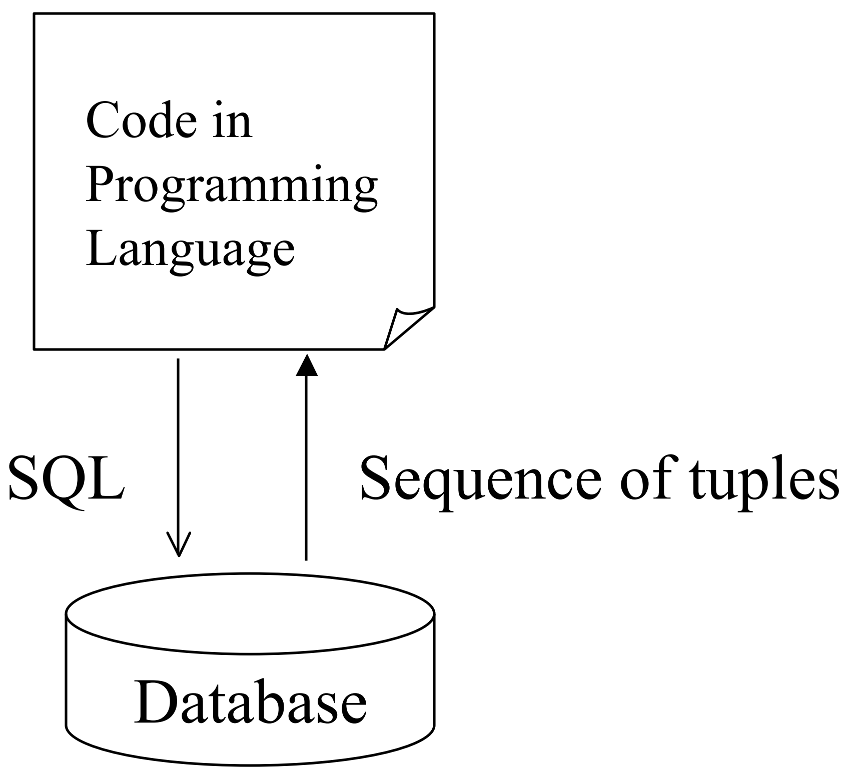
Database Programming
&
Transaction Management

Database Programming



Embedded SQL

- SQL commands can be called from within a host language (e.g., C or COBOL) program.
  - SQL statements can refer to host variables (including special variables used to return status).
  - Must include a statement to *connect* to the right database.
- SQL relations are (multi-) sets of records, with no *a priori* bound on the number of records. No such data structure in C.
  - SQL supports a mechanism called a *cursor* to handle this.

Variable Declaration

- Can use host-language variables in SQL statements
  - Must be prefixed by a colon (:)
  - Must be declared between

EXEC SQL BEGIN DECLARE SECTION

:

EXEC SQL END DECLARE SECTION

Variable Declaration in C

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

Embedded SQL

- All SQL statements embedded within a host program must be clearly marked.
- In C, SQL statements must be prefixed by EXEC SQL:

```
EXEC SQL
INSERT
INTO Sailors
VALUES(:c_sname,:c_sid,:c_rating,:c_age);
```

- Java embedding uses # SQL { …. };

SELECT - Retrieving Single Row

```
EXEC SQL SELECT S.sname, S.age
         INTO   :c_name, :c_age
         FROM   Sailors S
         WHERE S.sid = :c_sid;
```

SELECT - Retrieving Multiple Rows

- What if we want to embed the following query?

```
SELECT S.sname, S.age
FROM    Sailors
WHERE S.rating > :c_minrating
```

- Potentially, multiple rows will be retrieved
- How do we store the set of rows?
  - No equivalent data type in host languages like C

Cursors

- Can declare a cursor on a relation or query statement (which generates a relation).
- Can *open* a cursor, and repeatedly *fetch* a tuple then *move* the cursor, until all tuples have been retrieved.
  - Can use a special clause, called ORDER BY, in queries that are accessed through a cursor, to control the order in which tuples are returned.
    - Fields in ORDER BY clause must also appear in SELECT clause.
  - The ORDER BY clause, which orders answer tuples, is *only* allowed in the context of a cursor.
- Can also modify/delete tuple pointed to by a cursor.

Declaring a Cursor

- Cursor that gets names of sailors who've reserved a red boat, in alphabetical order

```
EXEC SQL DECLARE sinfo CURSOR FOR
        SELECT  S.sname
        FROM  Sailors S, Boats B, Reserves R
        WHERE  S.sid=R.sid AND
               R.bid=B.bid AND
               B.color='red'
        ORDER BY  S.sname
```

Opening/Fetching a Cursor

- To open the cursor (executed at run-time):
  - **OPEN sinfo;**
  - The cursor is initially positioned just before the first row
- To read the current row that the cursor is pointing to:
  - **FETCH sinfo INTO :c_name, :c_age**
- When FETCH is executed, the cursor is positioned to point at the next row
  - Can put the FETCH statement in a loop to retrieve multiple rows, one row at a time

Closing a Cursor

- When we're done with the cursor, we can close it:
  - CLOSE sinfo;
- We can re-open the cursor again. However, the rows retrieved might be different (depending on the value(s) of the associated variable(s) when cursor is opened)
  - Ex. If :c_minrating is now set to a different value, then the rows retrieved will be different

Embedding SQL in C: An Example

```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_minrating; float c_age;
EXEC SQL END DECLARE SECTION
c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
         SELECT S.sname, S.age
         FROM Sailors S
         WHERE S.rating > :c_minrating
         ORDER BY S.sname;
EXEC SQL OPEN sinfo;
do {
   EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
   printf("%s is %d years old\n", c_sname, c_age);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;
```

Dynamic SQL

- Allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```
char * sqlprog = "UPDATE account
                  SET balance = balance * 1.05
                  WHERE account_number = ?"
EXEC SQL PREPARE dynprog FROM :sqlprog;
char account [10] = "A-101";
EXEC SQL EXECUTE dynprog USING :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.

Database APIs: Alternative to embedding

Rather than modify compiler, add library with database calls (API)

- special standardized interface: procedures/objects
- passes SQL strings from language, presents result sets in a language-friendly way
- Microsoft's *ODBC* becoming C/C++ standard on Windows
- Sun's *JDBC* a Java equivalent
- Supposedly DBMS-neutral
  - a "driver" traps the calls and translates them into DBMS-specific code
  - database can be across a network

ODBC and JDBC

- 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
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java

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

SQL API in Java (JDBC)

```
Connection con = // connect
  DriverManager.getConnection(url, ”login", ”pass");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT name, rating FROM Sailors";
ResultSet rs = stmt.executeQuery(query);
try { // handle exceptions
   // loop through result tuples
   while (rs.next()) {
      String s = rs.getString(“name");
      Int n = rs.getFloat(“rating");
      System.out.println(s + "   " + n);
   }
} catch(SQLException ex) {
   System.out.println(ex.getMessage ()
      + ex.getSQLState () + ex.getErrorCode ());
}
```

Transaction Management - Overview

TRANSACTION MANAGEMENT

Airline Reservations	many updates
Statistical Abstract of the US	many queries

Atomicity – all or nothing principle

Serializability – the effect of transactions as if they occurred one at a time

Items – units of data to be controlled

- fine-grained – small items
- course-grained – large items
- (granularity)

Controlling access by locks

- Read – sharable with other readers shared
- Write – not sharable with anyone else exclusive

Model – (item, locktype, transaction ID)

Transactions

Transaction = a unit of work that must be:

1. *Atomic* = either all work is done, or none of it.
2. *Consistent* = relationships among values maintained.
3. *Isolated* = appear to have been executed when no other DB operations were being performed.
   – Often called *serializable* behavior.
4. *Durable* = effects are permanent even if system crashes.

Commit/Abort Decision

Each transaction ends with either:

1. *Commit* = the work of the transaction is installed in the database; previously its changes may be invisible to other transactions.
2. *Abort* = no changes by the transaction appear in the database; it is as if the transaction never occurred.
   - `ROLLBACK` is the term used in SQL and the Oracle system.

- In the ad-hoc query interface (*e.g.*, PostgreSQL psql interface), transactions are single queries or modification statements.
  - Oracle allows `SET TRANSACTION READ ONLY` to begin a multistatement transaction that doesn't change any data, but needs to see a consistent "snapshot" of the data.
- In program interfaces, transactions begin whenever the database is accessed, and end when either a `COMMIT` or `ROLLBACK` statement is executed.

Example

Sells(<u>bar</u>, <u>beer</u>, price)

- Joe's Bar sells Bud for $2.50 and Miller for $3.00.
- Sally is querying the database for the highest and lowest price Joe charges:

(1)
```
SELECT MAX(price) FROM Sells
WHERE bar = 'Joe''s Bar';
```
(2)
```
SELECT MIN(price) FROM Sells
WHERE bar = 'Joe''s Bar';
```

- At the same time, Joe has decided to replace Miller and Bud by Heineken at $3.50:

(3)
```
DELETE FROM Sells
WHERE bar = 'Joe''s Bar' AND
        (beer = 'Miller' OR beer = 'Bud');
```
(4)
```
INSERT INTO Sells
VALUES('Joe''s bar', 'Heineken', 3.50);
```

- If the order of statements is 1, 3, 4, 2, then it appears to Sally that Joe’s minimum price is greater than his maximum price.
- Fix the problem by grouping Sally’s two statements into one transaction, *e.g.*, with one SQL statement.

Example: Problem With Rollback

- Suppose Joe executes statement 4 (insert Heineken), but then, during the transaction thinks better of it and issues a `ROLLBACK` statement.
- If Sally is allowed to execute her statement 1 (find max) just before the rollback, she gets the answer $3.50, even though Joe doesn't sell any beer for $3.50.
- Fix by making statement 4 a transaction, or part of a transaction, so its effects cannot be seen by Sally unless there is a `COMMIT` action.

SQL Isolation Levels

Isolation levels determine what a transaction is allowed to see. The declaration, valid for one transaction, is:

`SET TRANSACTION ISOLATION LEVEL` *X*`;`

where:

- *X* = `SERIALIZABLE`: this transaction must execute as if at a point in time, where all other transactions occurred either completely before or completely after.
  - Example: Suppose Sally's statements 1 and 2 are one transaction and Joe's statements 3 and 4 are another transaction. If Sally's transaction runs at isolation level SERIALIZABLE, she would see the `Sells` relation either before or after statements 3 and 4 ran, but not in the middle.

- X = `READ COMMITTED`: this transaction can read only committed data.
  - Example: if transactions are as above, Sally could see the original `Sells` for statement 1 and the completely changed `Sells` for statement 2.
- X = `REPEATABLE READ`: if a transaction reads data twice, then what it saw the first time, it will see the second time (it may see more the second time).
  - Moreover, all data read at any time must be committed; *i.e.*, `REPEATABLE READ` is a strictly stronger condition than `READ COMMITTED`.
  - Example: If 1 is executed before 3, then 2 must see the Bud and Miller tuples when it computes the min, even if it executes after 3. But if 1 executes between 3 and 4, then 2 may see the Heineken tuple.

- X = `READ UNCOMMITTED`: essentially no constraint, even on reading data written and then removed by a rollback.
  - Example: 1 and 2 could see Heineken, even if Joe rolled back his transaction.

Independence of Isolation Levels

Isolation levels describe what a transaction *T* with that isolation level sees.

- They *do not* constrain what other transactions, perhaps at different isolation levels, can see of the work done by *T*.

Example

If transaction 3-4 (Joe) runs serializable, but transaction 1-2 (Sally) does not, then Sally might see `NULL` as the value for both min and max, since it could appear to Sally that her transaction ran between steps 3 and 4.

Authorization in SQL

- File systems identify certain access privileges on files, *e.g.*, read, write, execute.
- In partial analogy, SQL identifies six access privileges on relations, of which the most important are:

1. `SELECT` = the right to query the relation.
2. `INSERT` = the right to insert tuples into the relation – may refer to one attribute, in which case the privilege is to specify only one column of the inserted tuple.
3. `DELETE` = the right to delete tuples from the relation.
4. `UPDATE` = the right to update tuples of the relation – may refer to one attribute.

Granting Privileges

- You have all possible privileges to the relations you create.
- You may grant privileges to any user if you have those privileges "with grant option."
  - You have this option to your own relations.

Example

1. Here, Sally can query `Sells` and can change prices, but cannot pass on this power:

```
GRANT SELECT ON Sells,
      UPDATE(price) ON Sells
TO sally;
```

2. Here, Sally can also pass these privileges to whom she chooses:

```
GRANT SELECT ON Sells,
      UPDATE(price) ON Sells
TO sally
WITH GRANT OPTION;
```

Revoking Privileges

- Your privileges can be revoked.
- Syntax is like granting, but `REVOKE ... FROM` instead of `GRANT ... TO`.
- Determining whether or not you have a privilege is tricky, involving "grant diagrams" as in text. However, the basic principles are:

a) If you have been given a privilege by several different people, then all of them have to revoke in order for you to lose the privilege.

b) Revocation is transitive. if *A* granted *P* to *B*, who then granted *P* to *C*, and then *A* revokes *P* from *B*, it is as if *B* also revoked *P* from *C*.