COMP3311 Week 08 Lecture

Overview of Database Programming

DB Interface (in PHP)

Standard pattern for extracting data from DB:

```
$db = dbConnect("dbname=myDB");
...
$min = ...;
$q = "select a,b from R where c >= %d";
$r = dbQuery($db, mkSQL($q, $min));
while ($t = dbNext($r)) {
        list($a,$b) = $t;
        $sum = $a + $b;
        # or ...
        $sum = $t["a"] + $t["b"];
        # or ...
        $sum = $t[0] + $t[1];
        printf("Sum is %d\n", $sum);
}
...
```

PL access to DBs

In accessing DBs from PLs, your code contains:

- code in a programming language
- SQL query/update statements
- code to map between tuples and PL data

When you write PHP/DB scripts for web pages:

- also get HTML, JavaScript, Cascading Style Sheet (CSS), ...

Five different notations in one script!

... PL access to DBs

Trend in building enterprise systems over the last two decades:

- work in a single language (Java)
- implement everything in terms of objects

Approaches to achieving this:

- Enterprise Java Beans (EJBs) and Data Access Objects (DAOs)
- Object-relational mapping (e.g. Active Record, Hibernate)

Outcomes:

- more productive programmers, inefficient systems

... PL access to DBs

EJB Data Access Objects:

- business objects are represented by a collection of values
- values may be spread across multiple tables
- implement a business object class with operations
  - `create()` ... inserts new tuple(s), given object values
  - `getData()` ... fetch value of (typically) one attribute
  - `setData()` ... update value of (typically) one attribute
- essentially an OO wrapper around the SQL schema

... PL access to DBs

Active Record design pattern:

- treats tuples as core objects
- requires user to follow conventions in defining tables
- uses DBMS metadata to derive access methods
- provides access to DB via objects, no SQL needed
- automatically generates methods to access tables

Hibernate-style object-relational mapping:

- more like EJB "business objects"
- write an XML description of Objects-DB mapping
- system automatically produces SQL to access DB
- lazy loading avoids grabbing all data at once

Catalogs

Catalogs

An RDBMS maintains a collection of relation instances.

To do this, it also needs information *about* relations:

- name, owner, primary key of each relation
- name, data type, constraints for each attribute
- authorisation for operations on each relation

Similarly for other DBMS objects (e.g. views, functions, triggers, ...)

This information is stored in the *system catalog*.

(The "system catalog" is also called "data dictionary" or "system view")

... Catalogs

DBMSs use a hierarchy of namespaces to manage names:

Database (or **Catalog**)

- top-level namespace; contains schemas
- users connect to/work in a "current" database

Schema

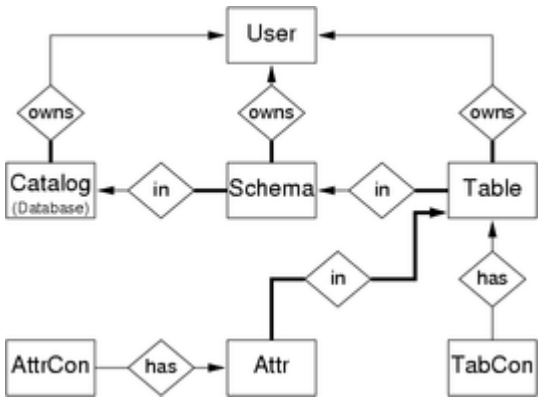
- second-level namespace; contains tables, views, etc.
- users typically work within a "current" schema

Table

- lowest-level namespace; contains attributes
- SELECT queries set a context for attribute names

... Catalogs

DBMSs store the catalog data in a collection of special tables:



(A small fragment of the meta-data tables in a typical RDBMS)

... Catalogs

SQL:2003 standard metadata: `INFORMATION_SCHEMA`.

`INFORMATION_SCHEMA` is available globally and includes:

Schemata(catalog_name, schema_name, schema_owner, ...)

Tables(table_catalog, table_schema, table_name, table_type, ...)

Columns(table_catalog, table_schema, table_name, column_name,
ordinal_position, column_default, data_type, ...)

Views(table_catalog, table_schema, table_name, view_definition, ...)

Table_Constraints(..., constraint_name, ..., constraint_type, ...)

etc. etc. etc.

PostgreSQL and Schemas

When you run `psql`

- you specify database (e.g., a2)
- it specifies a default schema: `public`

All object references are relative to the current schema.

Can access objects in other schemas via: *Schema*`.`*Object*

(e.g. `public.students`, `information_schema.tables`, ...)

To change schemas: **`set schema '`***Schema*`'`

PostgreSQL provides SQL standard `INFORMATION_SCHEMA`

Exercise: Exploring the Catalog (1)

Using the `INFORMATION_SCHEMA`, write a view to list the names of all tables in the `public` schema

The view should be defined/invoked as follows:

```
select * from myTables;
  name
--------
 table1
 table2
 table3
 ...
```

[Solutions]

Exercise: Exploring the Catalog (2)

Using the `INFORMATION_SCHEMA`, write a view to list the tables+attributes in

the public schema

The view should be defined/invoked as follows:

```
select * from mySchema;
 table  |            attributes
--------+---------------------------------
 table1 | attr1a, attr1b, attr1c, attr1d
 table2 | attr2a, attr2b
 table3 | attr3a, attr3b, attr3c
 ...
```

[Solutions]

Exercise: Exploring the Catalog (3)

15/32

Using the INFORMATION_SCHEMA, write a PLpgSQL function

```
create or replace function
    myTableDef(_table text) returns text
as ...
```

- whose argument is a table name (from the public schema)
- whose result is a CREATE TABLE statement to build the table
- only handle constraints mentioned in the columns table

Extension: add all constraints (not just the ones in columns)

[Solutions]

Exercise: Size of each Table

16/32

Write a PLpgSQL function to produce a list of user tables, along with a count of the number of tuples in each table.

The function should be defined/invoked as follows:

```
create type PopulationRecord as
    ("table" text, ntuples integer);

create or replace function
    dbpop() returns setof PopulationRecord
...
select * from dbpop();

 table | ntuples
-------+--------
     R |     25
     S |     12
```

[Solutions]

Exercise: Test-bed for Views

Write a PLpgSQL function that checks

- the results of invoking a specific view
- against a table of expected results
- along with reasonable error-checking (e.g. view does not exist)

Assume that:

- the view to be checked is called `V`
- the function is called `checkV()`
- it returns: correct, too many tuples, too few tuples
- a table `V_expected` contains results of `V`

[Solutions]

PostgreSQL Catalog

Most DBMSs had defined their own catalog tables before `INFORMATION_SCHEMA` was standardised.

The PostgreSQL catalog contains around 80 tables and views

- most describe schema data (tables, attributes, constraints, ...)
- others deal with DB configuration and statistics
- others deal with users, roles, privileges
- all are called `pg_`*XXX* (e.g. `pg_tables`)
- many have primary key via implicit `oid` attribute

This week's lecture "PostgreSQL Catalog" contains details of important catalog tables.

For full details, see Chapter 50 (Systems Catalog) in PostgreSQL documentation.

Security, Privilege, Authorisation

Database Access Control

Access to DBMSs involves two aspects:

- having execute permission for a DBMS client (e.g. `psql`)

- having a username/password registered in the DBMS

Establishing a *connection* to the database:

- user supplies database/username/password to client
- client passes these to server, which validates them
- if valid, user is "logged in" to the specified database

... Database Access Control

Note: we don't need to supply username/password to `psql`

- `psql` works out which user by who ran the client process
- we're all PostgreSQL super-users on our own servers
- servers are configured to allow super-user direct access

Note: access to databases via the Web involves:

- running a script on a Web server
- using the Web server's access rights on the DBMS

Access specified in `/srvr/YOU/pgsql/data/pg_hba.conf`

... Database Access Control

SQL standard doesn't specify details of users/groups/roles.

Some typical operations on users:

```
CREATE USER Name IDENTIFIED BY 'Password'
ALTER USER Name IDENTIFIED BY 'NewPassword'
ALTER USER Name WITH Capabilities
ALTER USER Name SET ConfigParameter = ...
```

Capabilities: super user, create databases, create users, etc.

Config parameters: resource usage, session settings, etc.

... Database Access Control

A user may be associated with a *group* (aka *role*)

Some typical operations on groups:

```
CREATE GROUP Name
ALTER GROUP Name ADD USER User1, User2, ...
ALTER GROUP Name DROP USER User1, User2, ...
```

Examples of groups/roles:

- AcademicStaff ... has privileges to read/modify marks
- OfficeStaff ... has privilege to read all marks
- Student ... has privilege to read own marks only

Database Access Control in PostgreSQL

In older versions of PostgreSQL ...

- USERs and GROUPs were distinct kinds of objects
- USERs were added via `CREATE USER` *UserName*
- GROUPs were added via `CREATE GROUP` *GroupName*
- GROUPs were built via `ALTER GROUP ... ADD USER ...`

In recent versions, USERs and GROUPs are unified by ROLEs

Older syntax is retained for backward compatibility.

... Database Access Control in PostgreSQL

PostgreSQL has two ways to create users ...

From the Unix command line, via the command

`createuser` *Name*

From SQL, via the statement:

```
CREATE ROLE UserName Options
-- where Options include ...
PASSWORD 'Password'
CREATEDB | NOCREATEDB
CREATEUSER | NOCREATEUSER
IN GROUP GroupName
VALID UNTIL 'TimeStamp'
```

... Database Access Control in PostgreSQL

Groups are created as ROLEs via

```
CREATE ROLE GroupName
--or--
CREATE ROLE GroupName WITH USER User1, User2, ...
```

and may be subsequently modified by

```
GRANT GroupName TO User1, User2, ...
REVOKE GroupName FROM User1, User2, ...
GRANT Privileges ... TO GroupName
REVOKE Privileges ... FROM GroupName
```

SQL Access Control

SQL access control deals with

- privileges on database objects (e.g. tables, view, functions, ...)
- allocating such privileges to roles (i.e. users and groups)

The user who creates an object is automatically assigned:

- ownership of that object
- a privilege to modify (ALTER) the object
- a privilege to remove (DROP) the object
- along with all other privileges specified below

... SQL Access Control

The owner of an object can assign privileges on that object to other users.

Accomplished via the command:

```
GRANT Privileges ON Object
TO ( ListOfRoles | PUBLIC )
[ WITH GRANT OPTION ]
```

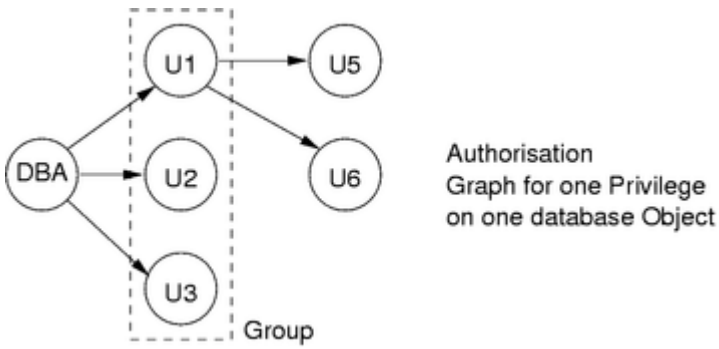
Privileges can be ALL (giving everything but ALTER and DROP)

WITH GRANT OPTION allows a user who has been granted a privilege to pass the privilege on to any other user.

... SQL Access Control

Effects of privilege granting

- are sometimes subtle (possible conflicts?)
- can be represented by an *authorisation graph*



... SQL Access Control

Privileges can be withdrawn via the command:

```
REVOKE Privileges ON Object
FROM ListOf (Users|Roles) | PUBLIC
CASCADE | RESTRICT
```

Normally withdraws Privileges from just specified users/roles.

`CASCADE` ... also withdraws from users they had granted to.

E.g. revoking from U1 also revokes U5 and U6

`RESTRICT` ... fails if users had granted privileges to others.

E.g. revoking from U1 fails, revoking U5 or U2 succeeds

... SQL Access Control

Privileges available for users on database objects:

`SELECT`:

- user can read all rows and columns of table/view
- this includes columns added later via `ALTER TABLE`

`INSERT` or `INSERT(`*ColName*`)`:

- user can insert rows into table
- if *ColName* specified, can only set value of that column

... SQL Access Control

More privileges available for users on database objects:

- `UPDATE`: user can modify values stored in the table
- `UPDATE(`*ColName*`)`: user can update specified column
- `DELETE`: user can delete rows from the table
- `REFERENCES(`*ColName*`)`: user can use column as foreign key
- `EXECUTE`: user can execute the specified function
- `TRIGGER`: user is allowed to create triggers on table

Produced: 22 April 2018