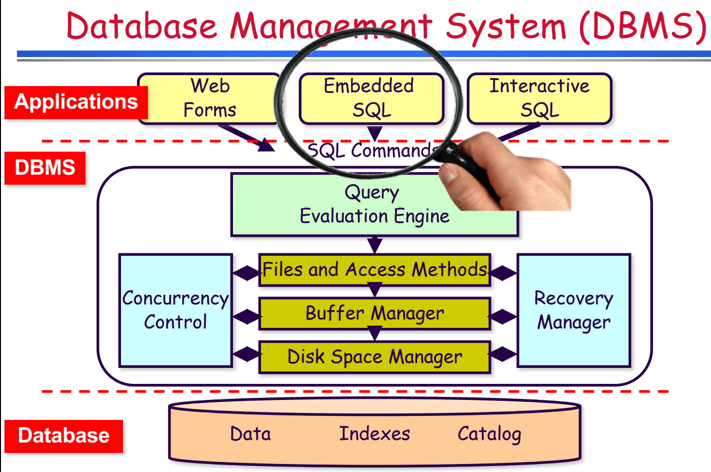
Database Programming at Large

Stored Procedures and Embedded SQL

Database Management System (DBMS)



Database Programming

- Objective:
  - To access a database from an **application** program (as opposed to interactive interfaces)
- Why?
  - An interactive interface is convenient but <u>not sufficient</u>
    - A majority of database operations are made thru application programs (increasingly thru web applications)

Database Programming Approaches

- Embedded commands:
  - Database commands are **embedded** in a general-purpose programming language
- Library of database functions:
  - Available to the host language for database calls; known as an **API** (Application Program Interface)
  - *e.g., JDBC, ODBC, PHP, Python*
- A brand new, full-fledged language
  - e.g., Oracle PL/SQL, Postgres PL/pgSQL
  - **P**rocedural **L**anguage extensions to SQL

Approach 3: SQL/PL

- Functions/procedures can be written in SQL itself, or in an external programming language
- Functions are very useful with specialized data types
  - E.g. functions to check if polygons overlap, or to compare images for similarity
- Some databases support **table-valued functions**, which can return a relation as a result
- SQL3 also supports a rich set of imperative constructs
  - Loops, if-then-else, case, assignment + exception handling
  - Similar to CSH script language
- Many DBMS have proprietary procedural extensions to SQL that differ from SQL3.

SQL Functions

authors (author, title, author_order)

- Definition of a Function

```
create or replace function author_count (name varchar(20))
    return integer
a_count integer;            -- local variable declaration
begin
    select count(author) into a_count    -- into is a tuple assignment operator
    from authors
    where authors.title=name;
    return a_count;
end;
/
```

- '/': Executes a PL/SQL block
- Invocation ?

```
SELECT title
FROM books4
WHERE author_count(title)> 1;
```

SQL Procedures

- Definition of a procedure:

```
create or replace procedure author_count_proc (in title varchar(20),
                                               out a_count integer )
    begin
        select count(author) into a_count
        from authors
        where authors.title = title;
    end;
    /
```

- Parameters Options: **IN, OUT, INOUT**
  - **Oracle syntax:** (title **in** varchar(20), a_count **out** integer)
- Invocation ?

ANSI SQL Procedures: Invocation

- Procedures can be invoked either within a trigger, an SQL procedure, or from embedded SQL, using the **Call** statement.
- E.g., from an SQL procedure block

```
declare a_count integer;
begin
call author_count_proc(`Database Systems', a_count);
end;
```

- SQL3 allows name **overloading** for function and procedures, as long as the number or types of arguments is different.

ANSI SQL: Procedures in Triggers

```
CREATE OR REPLACE TRIGGER Update_ShipDate
    AFTER INSERT OR UPDATE OF ShipDate
    ON Orders
    FOR EACH ROW
    BEGIN ATOMIC
        CALL UpdateShipDate(:new)
    END;
/
```

Oracle PL/SQL Procedure Invocation

❑ two ways to execute a procedure.

1) From the SQL prompt:

```
EXECUTE [or EXEC] procedure_name;
```

2) Within another procedure – simply use the procedure name:

```
procedure_name;
```

SQL*PLUS: Execute a PL/SQL Block

- ❑ To execute a PL/SQL block (procedure, trigger etc.), its "End;" should be followed by either
  - a slash '/': execute/process without showing the content of the SQL buffer
  - **run**: first shows the content of the SQL buffer and then executes it.
- ❑ Note that the dot (**.**), if entered as first character on the line ends inputting lines to the SQL buffer, without executing its content
- ❑ "show errors": List all the errors of latest SQL invocation
  also: "show errors trigger <name of trigger>

PL/pgSQL Function

❑ Create a function statement

```
CREATE FUNCTION func_name(…) RETURNS r_type AS
$$
[ DECLARE
  declarations ]
BEGIN
    statements
END;
$$ LANGUAGE plpgsql;
```

❑ Drop a function statement

```
DROP FUNCTION [IF EXISTS] func_name() [CASCADE|RESTRICT];
```

PL/pgSQL Example Function

```
create or replace function author_count (name varchar(20))
      returns integer as
 $$
declare
a_count integer;              -- local variable declaration
begin
    select count(author) into a_count
     from authors
    where authors.title=name;
    return a_count;
 end;
 $$ LANGUAGE plpgsql;
```

Trigger example in Postgres

```
CREATE TRIGGER Name_Trim
BEFORE INSERT
ON Student
FOR EACH ROW
  EXECUTE PROCEDURE trim_name();
```

PL/pgSQL Trigger Function

```
CREATE OR REPLACE FUNCTION trim_name()
    RETURNS trigger AS
$$
BEGIN
 NEW.name = LTRIM(NEW.name);
 RETURN NEW;
END;
$$
 LANGUAGE 'plpgsql';
```

More on triggers in Postgres

- CREATE [CONSTRAINT] TRIGGER *trig_name time event*

```
     ON table_name
      [ NOT DEFERRABLE l [ DEFERRABLE ]
        { INITIALLY IMMEDIATE l INITIALLY DEFERRED } ]
      [ FOR EACH { ROW | STATEMENT } ]
      [ WHEN ( condition ) ]
      EXECUTE PROCEDURE func_name ();
```

- Constraint triggers must be AFTER ROW triggers.
- SET CONSTRAINT trig_name < Evaluation Mode>

Oracle PL/SQL

- Based on ADA
- Assignments: direct (:=) and retrieval (INTO)
- Conditional Statements:

PL/pgSQL: Same but no { }

```
IF <condition>
THEN
        {statement;}
ELSE
        {<statement;>}
END IF;
```

```
IF <condition>
THEN
        {<statement;>}
 ELSIF <condtion>
 THEN
         {<statement;>}
  ELSE
        {<statement;>}
  END IF;
```

Oracle PL/SQL…

PL/pgSQL: Same but no { }

- Iterative Statements
  - Simple Loop

```
LOOP
   {<statement;>}
   EXIT; [or EXIT WHEN condition;]
 END LOOP;
```

  - While Loop

```
WHILE <condition> LOOP {<statement;>} END LOOP;
```

  - For Loop

```
FOR counter IN val1..val2
     {<statement;}
END LOOP;
```

PL/pgSQL: additional variants, e.g., `IN [REVERSE]`

Procedural Constructs: Exceptions

- Signaling of exception conditions, and declaring handlers for exceptions

  declare *out_of_stock* **condition**
  declare *exit* **handler for** *out_of_stock*
  begin
  …
  .. **signal** out_of_stock
  end

- The handler here is **exit** -- causes enclosing begin..end to be exited
- Other actions possible on exception

Exception handling in Oracle PL/pgSQL

- EXCEPTION clause at the end of a block:

```
BEGIN
  statements
   …
EXCEPTION
   WHEN condition [ OR condition … ] THEN
        handler_statements
   [ WHEN condition [ OR condition … ] THEN
        handler_statements
   ]
        …
   [ WHEN OTHER THEN handler_statements ]
 END;
```

Oracle PL/SQL: General Structure

```
Declare                                   -- optional
  x integer := 0;
  y student.sid%type;
  bad_data exception;
Begin                                     -- mandatory
  select count(*) into x
  from STUDENT
  where major = 'CS' ;
  if x < 1 then RAISE bad_data;
  else dbms_output.put_line ("Number of CS Majors =" || x);
  end if;
Exception                                 -- optional
  when bad_data then
    dbms_output.put_line ("troubles");
End;
```

Oracle PL/SQL: Var & Const

- **DECLARE**: introduces variables, constrants & records
- **Variables & Constants**
  - <variable_name> datatype [NOT NULL := value];
  - <constant_name> CONSTANT datatype := VALUE;
- Declaration of variables/constants based on a column from database table
  - <variable_name> table_name.column_name%type;
- E.g., y student.sid%type;

Oracle PL/SQL: Records

- **Record type**

```
TYPE <record_type_name> IS RECORD
 (<1st_col_name> datatype,
  <2nd_col_name> datatype, ...);
```

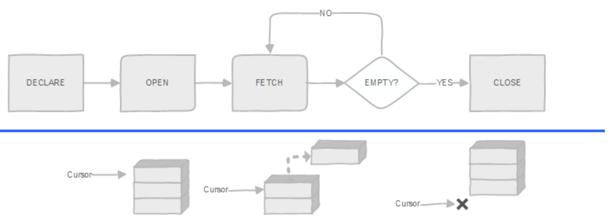
  - Declare fields based on a column from database table
    - col_name table_name.column_name%type;
- **Record variable declaration**
  - User-defined: record_name record_type_name;
  - DB-based: record_name table_name%ROWTYPE;
  - E.g., student_rec Student%rowtype;

PL/pgSQL

- Var & Const same as Oracle PL/SQL
- `rec_name RECORD;`
  - Has no predefined structure
  - Substructure is set when it is assigned a value
- `rec_name table_name%ROWTYPE;`
  - Structured to match the schema of *table_name*

Cursors: Multiple Tuple Retrieval

- If more than one tuples can be selected, then tuples must be processed one at a time by means of a cursor
  - This is similar to the record-at-a-time processing
- A cursor is a "pointer" to a tuple in a result of a query
  - Current tuple w.r.t. a cursor is the tuple pointed by the cursor



Source: http://www.postgresqltutorial.com/plpgsql-cursor/

Oracle PL/SQL Cursors

- CURSOR <cursor_name> IS <query>
  - It declares a cursor by defining a query to be associated with a cursor with it
- OPEN <cursor_name> brings the query result from the DB and positions the cursor before the first tuple
- CLOSE <cursor_name> closes the named cursor and deletes the associated result table
- Fetch copies into variables the current tuple and advances the cursor

  FETCH <cursor-name> INTO <record_name>;
  - `FETCH curs1 INTO rowvar;`

  FETCH <cursor-name> INTO <variable-list>;
  - `FETCH curs1 INTO foo, bar, baz;`

Oracle PL/SQL Cursor Retrieval

- Explicit Cursor Attributes

<cursor_name>%FOUND	- TRUE if tuple is returned
<cursor_name>%NOTFOUND	- TRUE if no tuple is returned
<cursor_name>%ROWCOUNT	- # tuple returned
<cursor_name>%ISOPEN	- TRUE if cursor is opened

PL/SQL Cursor Retrieval Example

Student(<u>SID</u>,Name,Major,QPA)

```
DECLARE
  CURSOR st_cursor  IS
      SELECT SID, Name, Major, QPA
      FROM Student;
  student_rec Student%rowtype;
BEGIN
   IF NOT st_cursor%ISOPEN
      THEN OPEN st_cursor;
   END IF;
   LOOP
      FETCH st_cursor INTO student_rec;
      EXIT WHEN st_cursor%NOTFOUND;
      dbms_output.put_line(student_rec.SID || ' ' || student_rec.Name || ' ' || student_rec.QPA);
   END LOOP;
   close st_cursor;
END;
```

Oracle PL/SQL Loop Cursor

```
CREATE OR REPLACE PROCEDURE proc_confirm_cost
AS
   CURSOR reservation_cursor IS
      SELECT *
      FROM Reservation;
BEGIN
   -- Loop across all reservation numbers & prints them out
   FOR reservation_record IN reservation_cursor
   LOOP
     dbms_output.put_line (reservation_record.Reservation_Number);
   END LOOP;
END;
/
```

Cursors in Postgres

- Declare cursor

```
cur_name [ [ NO ] SCROLL ] CURSOR [ ( args ) ] FOR query;
cur_name [ [ NO ] SCROLL ] CURSOR [ ( args ) ] IS query;
```

- E.g.,
  - `curs1 CURSOR FOR SELECT * FROM table1;`
  - `curs2 CURSOR (key integer) IS`
    `SELECT * FROM table1 WHERE att1 = key;`
- Before a cursor can be used, it must be opened
  - `OPEN curs1;`
  - `OPEN curs2(42);`
  - `OPEN curs2(key:=42);`

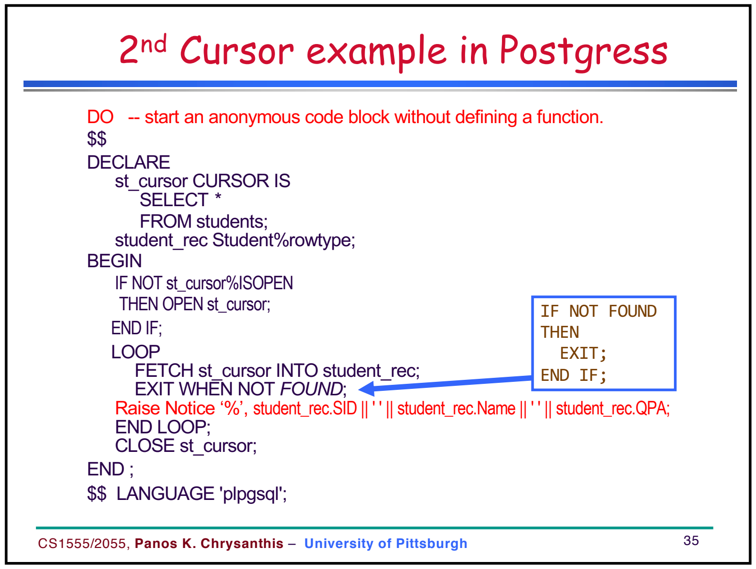
FETCHing & MOVEing in Postgres

- `FETCH [direction { FROM | IN }] cursor INTO target;`
  - *target* should be a RECORD or list of variables
  - RECORD can be a specific ROWTYPE
  - *direction* can take on many forms, e.g.:
    - `FETCH curs1 INTO rowvar;`
    - `FETCH curs2 INTO foo, bar, baz;`
    - `FETCH LAST FROM curs3 INTO x, y;`
    - `FETCH RELATIVE -2 FROM curs4 INTO recvar;`
  - Special variable **FOUND** will be set to true if a row is returned from the fetch
- `MOVE [ direction { FROM | IN } ] cursor;`
  - MOVE *direction* has all the flexibility of FETCH *direction*

Cursor example in Postgress

```
CREATE FUNCTION gpa_summer() RETURNS INTEGER AS $$
DECLARE
        gpa_sum INTEGER := 0;
        st_cursor CURSOR FOR
           SELECT ID, Name, Major, GPA FROM Students;
        student_rec Students%ROWTYPE;
BEGIN
        OPEN st_cursor;
        LOOP
          FETCH st_cursor INTO student_rec;
          IF NOT FOUND
          THEN
             EXIT;
          END IF;
          gpa_sum := gpa_sum + student_rec.GPA;
        END LOOP;
        CLOSE st_cursor;
        RETURN gpa_sum;
END;
$$ LANGUAGE plpgsql;
```

st_cursor% is not needed in plpgsql



2nd Cursor example in Postgress

```
DO   -- start an anonymous code block without defining a function.
$$
DECLARE
     st_cursor CURSOR IS
         SELECT *
         FROM students;
     student_rec Student%rowtype;
BEGIN
     IF NOT st_cursor%ISOPEN
      THEN OPEN st_cursor;
    END IF;
    LOOP
        FETCH st_cursor INTO student_rec;
        EXIT WHEN NOT FOUND;
     Raise Notice '%', student_rec.SID || ' ' || student_rec.Name || ' ' || student_rec.QPA;
     END LOOP;
     CLOSE st_cursor;
END ;
$$  LANGUAGE 'plpgsql';
```

```
IF NOT FOUND
THEN
  EXIT;
END IF;
```