


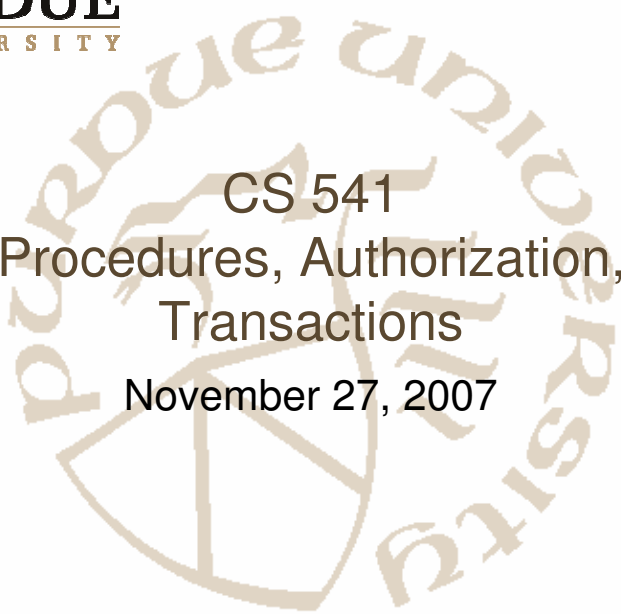
**PURDUE**  
UNIVERSITY

CS 541  
Procedures, Authorization,  
Transactions  
November 27, 2007

Fall 2007

Chris Clifton - CS541

Indiana  
Center for  
Database  
Systems



## PL/SQL

- Oracle's version of PSM (Persistent, Stored Modules).
  - Use via `sqlplus`.
- A compromise between completely procedural programming and SQL's very high-level, but limited statements.
- Allows local variables, loops, procedures, examination of relations one tuple at a time.
- Rough form:

```
DECLARE
    declarations
BEGIN
    executable statements
END ;
.
run ;
```
- `DECLARE` portion is optional.
- Dot and `run` (or a slash in place of `run ;`) are needed to end the statement and execute it.

## Simplest Form: Sequence of Modifications

```
Likes(drinker, beer)

BEGIN
  INSERT INTO Likes
    VALUES('Sally', 'Bud');
  DELETE FROM Likes
    WHERE drinker = 'Fred' AND
          beer = 'Miller';
END;
.
run;
```

## Procedures

Stored database objects that use a PL/SQL statement in their body.

### Procedure Declarations

```
CREATE OR REPLACE PROCEDURE
  <name>(<arglist>) AS
  <declarations>
  BEGIN
    <PL/SQL statements>
  END;
.
run;
```

- Argument list has name-mode-type triples.
  - Mode: IN, OUT, or IN OUT for read-only, write-only, read/write, respectively.
  - Types: standard SQL + generic types like NUMBER = any integer or real type.
  - Since types in procedures *must* match their types in the DB schema, you should generally use an expression of the form
 

```
relation.attribute %TYPE
```

 to capture the type correctly.

## Example

A procedure to take a beer and price and add it to Joe's menu.

```
Sells(bar, beer, price)

CREATE PROCEDURE joeMenu(
  b IN Sells.beer %TYPE,
  p IN Sells.price %TYPE
) AS
BEGIN
  INSERT INTO Sells
  VALUES('Joe''s Bar', b, p);
END;
```

- Note “run” only stores the procedure; it doesn't execute the procedure.

## Invoking Procedures

A procedure call may appear in the body of a PL/SQL statement.

- Example:

```
BEGIN
    joeMenu('Bud', 2.50);
    joeMenu('MooseDrool', 5.00);
END;

.
run;
```

## Assignment

Assign expressions to declared variables with  
:=.

### Branches

```
IF <condition> THEN
    <statement(s)>
ELSE
    <statement(s)>
END IF;
```

- But in nests, use ELSIF in place of ELSE IF.

### Loops

```
LOOP
    . . .
    EXIT WHEN <condition>
    . . .
END LOOP;
```

## Queries in PL/SQL

1. *Single-row selects* allow retrieval into a variable of the result of a query that is guaranteed to produce one tuple.
2. *Cursors* allow the retrieval of many tuples, with the cursor and a loop used to process each in turn.

## Single-Row Select

- Select-from-where in PL/SQL *must* have an INTO clause listing variables into which a tuple can be placed.
- It is an *error* if the select-from-where returns more than one tuple; you should have used a cursor.

### Example

- Find the price Joe charges for Bud (and drop it on the floor).

```
Sells(bar, beer, price)
DECLARE
  p Sells.price %TYPE;
BEGIN
  SELECT price
  INTO p
  FROM Sells
  WHERE bar = 'Joe's Bar' AND beer = 'Bud';
END;
.
```

```
run
```

## Cursors

Declare by:

```
CURSOR <name> IS  
  select-from-where statement
```

- Cursor gets each tuple from the relation produced by the select-from-where, in turn, using a *fetch statement* in a loop.
  - Fetch statement:

```
FETCH <cursor name> INTO  
  variable list ;
```
- Break the loop by a statement of the form:

```
EXIT WHEN <cursor name> %NOTFOUND ;
```

  - True when there are no more tuples to get.
- Open and close the cursor with `OPEN` and `CLOSE`.

## Example

A procedure that examines the menu for Joe's Bar and raises by \$1.00 all prices that are less than \$3.00.

```
Sells(bar, beer, price)
```

- This simple price-change algorithm can be implemented by a single `UPDATE` statement, but more complicated price changes could not.

```
CREATE PROCEDURE joeGouge() AS
  theBeer Sells.beer%TYPE;
  thePrice Sells.price%TYPE;
  CURSOR c IS
    SELECT beer, price
    FROM Sells
    WHERE bar = 'Joe''s bar';
BEGIN
  OPEN c;
  LOOP
    FETCH c INTO theBeer, thePrice;
    EXIT WHEN c%NOTFOUND;
    IF thePrice < 3.00 THEN
      UPDATE Sells
      SET price = thePrice + 1.00
      WHERE bar = 'Joe''s Bar'
      AND beer = theBeer;
    END IF;
  END LOOP;
  CLOSE c;
END;
.
```

run

## Row Types

Anything (e.g., cursors, table names) that has a tuple type can have its type captured with %ROWTYPE.

- We can create temporary variables that have tuple types and access their components with dot.
- Handy when we deal with tuples with many attributes.

## Example

The same procedure with a tuple variable bp.

```
CREATE PROCEDURE joeGouge() AS
  CURSOR c IS
    SELECT beer, price
    FROM Sells
    WHERE bar = 'Joe''s bar';
  bp c%ROWTYPE;
BEGIN
  OPEN c;
  LOOP
    FETCH c INTO bp;
    EXIT WHEN c%NOTFOUND;
    IF bp.price < 3.00 THEN
      UPDATE Sells
      SET price = bp.price + 1.00
      WHERE bar = 'Joe''s Bar'
      AND beer = bp.beer;
    END IF;
  END LOOP;
  CLOSE c;
END;
```

.  
run

## 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  $C$ .

### 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)**

## 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(bar, beer, 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:

- $X = \text{SERIALIZABLE}$ : 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.

- $X = \text{READ COMMITTED}$ : 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.
- $X = \text{REPEATABLE READ}$ : 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.

- $X = \text{READ UNCOMMITTED}$ : 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.