CSE 4409: Database Management Systems II

Abu Raihan Mostofa Kamal

Professor, CSE Department
Islamic University of Technology (IUT)
Gazipur-1704 Bangladesh
raihan.kamal@iut-dhaka.edu
+8801843925543

January 17, 2024

Chapter Outline

Introduction to PL SQL

- PL/SQL lets you **write code once** and deploy it in the **database nearest** the data. PL/SQL can simplify application development, optimize execution, and improve resource utilization in the database.

Introduction to PL SQL

- PL/SQL lets you **write code once** and deploy it in the **database nearest** the data. PL/SQL can simplify application development, optimize execution, and improve resource utilization in the database.
- The language is a **case-insensitive** programming language, like SQL.

Introduction to PL SQL

- PL/SQL lets you **write code once** and deploy it in the **database nearest** the data. PL/SQL can simplify application development, optimize execution, and improve resource utilization in the database.
- The language is a **case-insensitive** programming language, like SQL.
- **History:** PL/SQL was developed by modeling concepts of structured programming, static data typing, modularity, exception management, and parallel (concurrent) processing found in the **Ada programming language**. Similar to **Pascal**.

PL/SQL Block

- PL/SQL is a blocked programming language. A Block is the smallest working unit of the program code.

PL/SQL Block

- PL/SQL is a blocked programming language. A Block is the smallest working unit of the program code.
- Block may be either **anonymous** or **named**.

PL/SQL Block

- PL/SQL is a blocked programming language. A Block is the smallest working unit of the program code.
- Block may be either **anonymous** or **named**.
- Named block: Function, Procedure, Trigger, Package

PL/SQL Block

- PL/SQL is a blocked programming language. A Block is the smallest working unit of the program code.
- Block may be either **anonymous** or **named**.
- Named block: Function, Procedure, Trigger, Package
- **Anonymous block** is normally used for testing and debugging. To test a subprogram in a one-time processing activities it is mainly used.

PL/SQL Block: Example

```
1
2       [DECLARE]
3       declaration_statements
4       BEGIN
5       execution_statements
6       [EXCEPTION]
7       exception_handling_statements
8       END;
9       /
```

Note: No curly braces .., rather BEGIN .. END is used.

PL/SQL Block: Example (Cont.)

```
--this will do nothing

 BEGIN
NULL;
END;
/

 ---hello world code
 SET SERVEROUTPUT ON SIZE 1000000
 BEGIN
 dbms_output.put_line('Hello World.');
 END;
 /
```

Note: Use Single Quote for strings

PL/SQL Block: Example (Cont.)

```
--scanf--

DECLARE
my_var VARCHAR2(30);
BEGIN
my_var := '&i';
dbms_output.put_line('Hello '|| my_var );
END;
```

Note: In the area of Database such input program is **rarely** used.

Lexical delimiters: Symbols used

- := **Assignment**
  The assignment operator is a colon immediately followed by an equal symbol. It is the only assignment operator in the language. You assign a right operand to a left operand, like:
  a := b + c;

Lexical delimiters: Symbols used

- := Assignment
  The assignment operator is a colon immediately followed by an equal symbol. It is the only assignment operator in the language. You assign a right operand to a left operand, like:
  a := b + c;
- . Association
  The component selector is a period, and it glues references together, for example, a schema and a table, a package and a function.

```
1  schema_name.table_name
2  package_name.function_name
3  object_name.member_method_name
4  cursor_name.cursor_attribute
```

Lexical delimiters: Symbols used (Cont.)

- **@ Association.** The remote access indicator lets you access a remote database through database links.

Lexical delimiters: Symbols used (Cont.)

- **@ Association.** The remote access indicator lets you access a remote database through database links.
- **= Comparison.**

Lexical delimiters: Symbols used (Cont.)

- **@ Association.** The remote access indicator lets you access a remote database through database links.
- **= Comparison.**
- **<> != Comparison** Not equal to.

Lexical delimiters: Symbols used (Cont.)

- @ **Association.** The remote access indicator lets you access a remote database through database links.
- = **Comparison.**
- <> != **Comparison** Not equal to.
- – **Delimiter** In-line comment.

Lexical delimiters: Symbols used (Cont.)

- **@ Association.** The remote access indicator lets you access a remote database through database links.
- **= Comparison.**
- **<> != Comparison** Not equal to.
- **– Delimiter** In-line comment.
- **/* Delimiter */** Multi-line comment.

Lexical delimiters: Symbols used (Cont.)

- " **Delimiter.** Quoted identifier delimiter is a double quote. It lets you access tables created in case-sensitive fashion. For example, you create a case-sensitive table or column by using quoted identifier delimiters:

```
--create table in case-sensitive way
CREATE TABLE "Demo"
("Demo_ID" NUMBER
, demo_value VARCHAR2(10));
You insert a row by using the following quote-delimited syntax:

--insert value in it
INSERT INTO "Demo1" VALUES (1,'One Line ONLY.');
```

Data Types

- Number and Boolean
- String: varchar2(size)
- Date
- Large Object

Data Types (Cont.)

Boolean. The BOOLEAN datatype has three possible values: TRUE, FALSE, and NULL.

```
1  var1 BOOLEAN; -- Implicitly assigned a null value.
2  var2 BOOLEAN NOT NULL := TRUE; -- Explicitly assigned a TRUE value.
3  var3 BOOLEAN NOT NULL := FALSE; -- Explicitly assigned a FALSE value.
```

There is little need to **subtype** a BOOLEAN datatype, but you can do it. The subtyping syntax is:

```
SUBTYPE booked IS BOOLEAN;
room booked;
```

Data Types (Cont.)

- **Characters and Strings.** The VARCHAR2 datatype stores **variable-length** character strings. Maximum string length (in bytes or characters(?)) between 1 and **4000** bytes.
- Difference between char and varchar2: The following program illustrates it:

```
1   DECLARE
2   c CHAR(2000) := 'hello';
3   v VARCHAR2(32767) := 'hello';
4   BEGIN
5   dbms_output.put_line('The len of c is :'||LENGTH(c)||);
6   dbms_output.put_line('The len of v is :'||LENGTH(v)||);
7   END;
8   /
9   --OUTPUT
10  The len of c is :2000
11  The len of v is : 5
```

Data Types (Cont.)

- **More on VARCHAR2:** Globalization support allows the use of various character sets for the character datatypes. Globalization support lets you process **single-byte** and **multibyte** character data and convert between character sets.
- **Example:**

```
1    var1 VARCHAR2(100); -- Implicitly sized at 100 byte.
2    var2 VARCHAR2(100 BYTE); -- Explicitly sized at 100 byte.
3    var3 VARCHAR2(100 CHAR); -- Explicitly sized at 100 character.
```

Data Types (Cont.)

Datetime Datatypes: Here will will consider **only 2 variants** as such:

1. Date
2. TIMESTAMP

DATE Datatype

DATE is not same as Varchar2

The DATE datatype stores date and time information. Although date and time information can be represented in both character and number datatypes, the DATE datatype has **special associated properties**. For each DATE value, Oracle stores the following information: **century, year, month, date, hour, minute, and second**.

- Default format is 'YY-MON-DD'
- `to_char` and `to_date` built-ins are commonly used.

DATE: *TO_CHAR*

Whenever a DATE value is displayed, Oracle will call **TO_CHAR** automatically with the default DATE format. However, you may override the default behavior by calling **TO_CHAR** explicitly with your own DATE format. For example:

```
1  SELECT TO_CHAR(b, 'YYYY/MM/DD') AS b
2  FROM x;
3
4  --output--.
5
6  2021/12/05
```

DATE: *TO_CHAR* (Cont.)

The general usage of `TO_CHAR` is : `TO_CHAR(<date>, '<format>')`
Some of the more popular ones include:

Format	Meaning
MM	Numeric month (e.g., 07)
MON	Abbreviated month name (e.g., JUL)
MONTH	Full month name (e.g., JULY)
DD	Day of month (e.g., 24)
YYYY	4-digit year (e.g., 2021)
HH	Hour of day (1-12)
HH24	Hour of day (0-23)
MI	Minute (0-59)
SS	Second (0-59)

DATE: *TO_DATE*

You can use the `TO_DATE()` function to convert a string to a date before inserting:

```
1
2 INSERT INTO persons
3 (ID,name,joining_date)
4 VALUES    ( 105,'Mr. Test',
5 TO_DATE( 'December 01, 2021', 'MONTH DD, YYYY' ));
```

DATE: *TO_DATE* (Cont.)

***TO_DATE* in the where clause:** Instead of relying the default format, it is always safer to user `TO_DATE` to covert it to date explicitly.

Example Code: *Find the employees id,name who joined after 10:30 on December 1, 2021.*

```
SELECT ID,NAME
FROM EMP
WHERE JOINING_DATE>TO_DATE('DECEMBER 01,2021:10:30:00',
'MONTH DD,YYYY:HH24:MI:SS');

-- it can also be used in between clause in the same way
```

DATE: More..

- Date1-Date2 is the total number of days between them.
- Always use the built-in functions to manipulate date type data. Few commonly used built-ins are given below (see official documentation for further information)

Function	Example	Desc
`ADD_MONTHS`	`SELECT ADD_MONTHS( SYSDATE, 14 )` FROM DUAL	Add a number of months (n) to a date and return the same day which is n of months away.
`EXTRACT`	EXTRACT(MONTH FROM SYSDATE)	Extract a value of a date time field, legal values are YEAR, MONTH, DAY

Constants in PL/SQL

- CONSTANT keyword is used. The value must be initialized and can not be changed.

Example:

```
DECLARE
-- constant declaration
pi constant number := 3.141592654;
radius number(5,2);
area number (10, 2);

BEGIN
-- processing
radius := 12.65;
area := pi * radius * radius;
-- output

dbms_output.put_line('Area of the circle is : ' || area);
END;
```

Anchored Declarations

- Instead of "hardcoded" or explicit datatypes, it is sometimes very useful to anchor variables with other existing variables.
- PL/SQL offers two kinds of anchoring:
  i. **Scalar Anchoring.** %TYPE attribute is used to define a variable based on a tables column or some other PL/SQL scalar variable.
  ii. **Record Anchoring.** %ROWTYPE attribute is used to define a record structure based on a table or a predefined PL/SQL explicit cursor.

Anchored Declarations: Example

```
1        l_company_id company.company_id%TYPE;
```

Large Objects

Large Objects: Definition

Large Objects (**LOB**s) are a set of datatypes that are designed to hold large amounts of data. A LOB can hold up to a maximum size **ranging from 8 terabytes to 128 terabytes** depending on how your database is configured.

Large Objects: Why?

In the world today, applications must deal with the following kinds of data:

- **Simple and Complex data.** This data can be organized into simple tables that are structured based on business rules.Complex data includes Oracle database such as collections, references, and user-defined types.

Large Objects: Why?

In the world today, applications must deal with the following kinds of data:

- **Simple and Complex data.** This data can be organized into simple tables that are structured based on business rules.Complex data includes Oracle database such as collections, references, and user-defined types.
- **Semi-structured data.** It is not typically interpreted by the database.For example, an **XML document/word document** that is processed by your application or an external service, can be thought of as semi-structured data.

Large Objects: Why?

In the world today, applications must deal with the following kinds of data:

- **Simple and Complex data.** This data can be organized into simple tables that are structured based on business rules.Complex data includes Oracle database such as collections, references, and user-defined types.
- **Semi-structured data.** It is not typically interpreted by the database.For example, an **XML document/word document** that is processed by your application or an external service, can be thought of as semi-structured data.
- **Unstructured data.** This kind of data is not broken down into smaller logical structures and is not typically interpreted by the database or your application. A **photographic image** stored as a binary file is an example of unstructured data.

Large Objects: Why?

In the world today, applications must deal with the following kinds of data:

- **Simple and Complex data.** This data can be organized into simple tables that are structured based on business rules.Complex data includes Oracle database such as collections, references, and user-defined types.
- **Semi-structured data.** It is not typically interpreted by the database.For example, an **XML document/word document** that is processed by your application or an external service, can be thought of as semi-structured data.
- **Unstructured data.** This kind of data is not broken down into smaller logical structures and is not typically interpreted by the database or your application. A **photographic image** stored as a binary file is an example of unstructured data.

Large objects are **suitable** for : **semi-structured data and unstructured data**.