מעבדות מס' 9,10

אילוצים על ערכי שדות

ראינו אילוצי Keys ו-Foreign Keys. ראינו גם אילוץ NOT NULL. נראה כעת אילוצים בעזרת CHECK. בפרקטיקה, אילוץ CHECK קובע אנמרציה של ערכים או איזשהו אי שוויון אריתמטי. אך ככלל, ניתן לתת כל ביטוי, מסובך כמה שיהיה, שיכול להופיע לאחר WHERE בשאילתה. התנאי יכול להתייחס אפילו ל-Relation אחר המופיע ב-FROM של SubQuery.
התנאי נבדק בכל פעם ששורה מקבלת ערך חדש ל-attribute שעליו יש CHECK. אם ישנה הפרה, המודיפיקציה נדחית.
נראה דוגמאות:

```
CREATE TABLE Table3 (Num1 INTEGER,
                     Num2 INTEGER CHECK (Num2<70),
                     Ch VARCHAR(1) CHECK (Ch IN ('A', 'B')),
                     PRIMARY KEY (Num1, Num2),
                     FOREIGN KEY (Num1, Num2) REFERENCES Table1(Num1, Num2));
```

נראה דוגמה לתנאי שמתייחס ל-Relation אחר המופיע ב-FROM של SubQuery. בהגדרה הבסיסית של Student_In_Course, הגדרנו:

```
CREATE TABLE Student_in_Course (ssnumber NUMBER,
                                ccnumber NUMBER,
                                PRIMARY KEY (ssnumber, ccnumber),
                                FOREIGN KEY (ssnumber) REFERENCES Students (snumber),
                                FOREIGN KEY (ccnumber) REFERENCES Courses (cnumber));
```

ננסה "**לחקות**" את אילוץ ה-Foreign Key בעזרת CHECK:

```
CREATE TABLE Student_in_Course (ssnumber NUMBER,
                                ccnumber NUMBER CHECK (ccnumber IN (SELECT cnumber
                                FROM Courses)),
                                PRIMARY KEY (ssnumber, ccnumber),
                                FOREIGN KEY (ssnumber) REFERENCES Students (snumber));
```

האילוץ על ccnumber:

- ידאג לכך שכשנוסיף שורה בטבלה זו, אם הערך של ccnumber אינו מספר קורס בטבלת הקורסים, ההוספה תידחה.
- ידאג לכך שכשנעדכן ערך של ccnumber באחיד השורות, אם הערך אינו מספר קורס בטבלת הקורסים, העדכון יידחה.
- **אך**, מה יקרה כשנמחק שורה בטבלת Courses? מדוע?

לכן החיקוי לא הצליח.

כאן סיימנו את החומר ב-SQL. נעבור ל-PL/SQL

ראשי התיבות PL הם Procedural Language. PL/SQL מרחיב את SQL ומחזק את כוחו על ידי הענקת כלים משפות פרוצדוראליות. בלוק (תוכנית) ב-PL/SQL הוא בעל המבנה הבא:

```
DECLARE
/* חלק הצהרתי: משתנים, טיפוסים ופרוצדורות. */

BEGIN
/* חלק ביצועי: משפטים פרוצדוראליים ומשפטי SQL */
/* זהו החלק היחיד שנדרש בבלוק. (זהו החלק העיקרי שרץ). */

EXCEPTION
/* חלק הטיפול בחריגים. משפטי טיפול בטעויות יבואו כאן. */
```

משפטי ה-SQL שיכולים להשתלב בתוכנית PL/SQL הם אלו המתחילים ב-SELECT, UPDATE, DELETE. יש לדעת כי **תחביר ה-SELECT שונה מהרגיל** (נראה בהמשך). משפטי הגדרה כמו CREATE, DROP, ALTER **אינם** מותרים ב-PL/SQL.
הכלים הפרוצדוראליים ש-PL/SQL תומך בהם הם השמות, התניות, לולאות, קריאות לפרוצדורות והדקים (Triggers).
PL/SQL בדומה ל-SQL אינו רגיש לאותיות קטנות/גדולות. הערות (בדומה לשפת C, יבואו בצורה /* ..... */

כדי להריץ תוכנית PL/SQL עלינו להוסיף 2 שורות בסוף התוכנית, הראשונה רק עם נקודה (A line with single dot), ולאחריה שורה עם run; (הסימן נקודה פסיק לאחר המלה run). הרצת התוכנית תתבצע על ידי שמירתה בקובץ, ופנייה אליו בעזרת הסימן @ מ-SQLPLUS.

טיפוסי המשתנים

אינפורמציה בין תוכנית ה-PL/SQL לבין בסיס הנתונים מועברת דרך משתנים. טיפוסי המשתנים המותרים הם:

- אחד מהטיפוסים בהם אנו משתמשים ב-SQL לשדות.
- טיפוס כללי: NUMBER – יכול להחזיק מס' שלם או ממשי, או VARCHAR(n) – למחרוזת בגודל עד n.
- טיפוס המוגדר להיות **מפורשות** זהה לטיפוס שדה בטבלה.

דוגמה להגדרה:

```
DECLARE
        i       NUMBER;
        str     VARCHAR(15);
```

PL/SQL תומך בטיפוס BOOLEAN (שאינו טיפוס שדה חוקי ב-SQL).
במקרים רבים, משתנה משמש לצורך ביצוע מניפולציות על מידע שמאוחסן ב-Relation קיים. במקרה זה וכדי למנוע טעות בהגדרת המשתנה, מגדירים אותו בדיוק כטיפוס השדה על ידי הסימון %TYPE:

```
DECLARE
        FirstNum Table1.Num1%TYPE;
```

מגדיר משתנה ששמו FirstNum שהוא כטיפוס השדה Num1 ב-Table1.

משתנה יכול להיות מטיפוס רשומה עם כמה שדות. הדרך הפשוטה להגדרה היא על ידי %ROWTYPE:

```
DECLARE
        NumbersTuple Table1%ROWTYPE;
```

יוצר משתנה NumbersTuple להיות רשומה עם השדות Num1,Num2,Num3 (מתוך ידיעה שה-Relation ששמו Table1 הוא עם הסכימה Table1(Num1,Num2,Num3)).

ערכו ההתחלתי של כל משתנה, ללא תלות בטיפוסו, הוא NULL. ניתן להשים ערכים למשתנים על ידי האופרטור := (נקודותיים שווה). ניתן להשים ערך למשתנה גם בזמן הגדרה. כדוגמה:

```
DECLARE
        a NUMBER
BEGIN
        a := a + 1;
END;
.
run;
```

תוכנית זו, למעשה, כמעט לא עושה כלום.

נראה דוגמה נוספת לתוכנית:

```
CREATE TABLE T1(
   e INTEGER,
   f INTEGER
);

DELETE FROM T1;
INSERT INTO T1 VALUES(1, 3);
INSERT INTO T1 VALUES(2, 4);

/* Above is plain SQL; below is the PL/SQL program. */

DECLARE
   a NUMBER;
   b NUMBER;
BEGIN
   SELECT e,f INTO a,b FROM T1 WHERE e>1;
   INSERT INTO T1 VALUES(b,a);
END;
.
run;
```

הניואנס העיקרי הוא בצורה של פקודת ה-SELECT ב-PL/SQL (ששונה מהצורה ב-SQL). אחרי ה-SELECT ושדותיו, **חייבת** לבוא המלה INTO ורשימת משתנים, אחד לכל attribute של ה-SELECT, לשם ייכנסו מרכיבי **<u>השורה</u>** המוחזרת מה-SELECT. אגב, השורה המוחזרת ולא השורות המוחזרות, היות ו-SELECT ב-PL/SQL **תעבוד רק אם תוצאת השאילתה היא שורה בודדת**. במקרים שתוצאת השאילתה מחזירה יותר משורה בודדת, יש לעבוד עם Cursor (סמן, מצביע), כפי שיתואר בהמשך.

מה תעשה התוכנית שלעיל??? בדקו!!! **(הופ טרללה, גדלתם בשנה, וכבר אין תשובות לשאלות בדף).**

<u>התניות</u>

משפט IF נראה כך:

```
IF <condition> THEN <statement_list> ELSE <statement_list> END IF;
```

חלק ה-ELSE אינו הכרחי. תנאי מקונן יראה כך:

```
IF <condition_1> THEN ...
ELSIF <condition_2> THEN ...
... ...
ELSIF <condition_n> THEN ...
ELSE ...
END IF;
```

נראה דוגמה לשימוש ב-IF:

```
DECLARE
   a NUMBER;
   b NUMBER;
BEGIN
   SELECT e,f INTO a,b FROM T1 WHERE e>1;
   IF b=1 THEN
      INSERT INTO T1 VALUES(b,a);
   ELSE
      INSERT INTO T1 VALUES(b+10,a+10);
   END IF;
END;
.
run;
```

לולאות

לולאות נכתבות כך:

```
LOOP
   <loop_body> /* A list of statements. */
END LOOP;
```

כאשר לפחות שורה אחת ב-<loop_body> תכלול משפט יציאה EXIT מהצורה:

```
EXIT WHEN <condition>;
```

הלולאה תצא כאשר התנאי מתקיים, וכמובן במקום שנאמר לה לצאת. לדוגמה:

```
DECLARE
   i NUMBER := 1;
BEGIN
   LOOP
      INSERT INTO T1 VALUES(i,i);
      i := i+1;
      EXIT WHEN i>100;
   END LOOP;
END;
.
run;
```

הלולאה תכניס לטבלה T1 את הזוגות (1,1) עד (100,100).

לולאת WHILE היא מהצורה:

```
WHILE <condition> LOOP
      <loop_body>
END LOOP;
```

לולאת FOR היא מהצורה:

```
FOR <var> IN <start>..<finish> LOOP
      <loop_body>
END LOOP;
```

משתנה ה-<var> הוא לוקאלי ללולאה ולא צריך להיות מוגדר. <start> ו-<finish> הינם קבועים.

Cursor (מצביע, סמן)

Cursor הוא משתנה שרץ על שורות (tuples) ב-relation. ה-relation יכול להיות טבלה או אפילו פלט של תשאול מבט. אנו מעבירים אל ה-Cursor כל שורה ויכולים לעבד את הערכים בשורה זו. אם ה-relation מאוחסן פיזית, נוכל גם לעדכן או למחוק שורה שעליה עומד ה-Cursor.

התוכנית הבאה תמחק מ-T1 הנדון את כל השורות בהם ערך השדה הראשון קטן מערך השדה השני, ותכניס את השורה ההפוכה ל-T1.

מספר הערות בקשר לתוכנית:

- בשורות (2) ו-(3) העדפנו להקל על עצמנו בהגדרת טיפוסי המשתנים ולהיות יותר זהירים.
- בשורות (4) עד (8) מופיעה הגדרת ה-Cursor ששמו T1Cursor. בשורה (8) הגדרת ה-Cursor היא FOR UPDATE היות ואנו מעדכנים את T1 בשורה (14) בעזרת ה-Cursor.
- בשורה (10) אנו פותחים את ה-Cursor – שלב הכרחי.
- בשורה (12) אנו **מביאים** דרך ה-Cursor למשתנים לוקאליים. ככלל, משפט FETCH צריך לספק משתנים לכל מרכיבי השורה שאותה אנו מקבלים. היות והשאילתה בשורות (5) עד (7) מייצרת זוגות, אנו סיפקנו שני משתנים, ולא פחות חשוב, בטיפוסם המתאים.
- בשורה (13) בדיקת סיום הלולאה. NOTFOUND% לאחר שם של Cursor נכון רק כאשר פעולת FETCH עם Cursor זה נכשלה למצוא עוד שורות.
- בשורה (14) המחיקה בעזרת התנאי המיוחד ב-WHERE שהוא CURRENT OF T1Cursor.
- בשורה (17) אנו סוגרים את ה-Cursor.

```
 1) DECLARE
      /* Output variables to hold the result of the query: */
 2)    a T1.e%TYPE;
 3)    b T1.f%TYPE;
      /* Cursor declaration: */
 4)    CURSOR T1Cursor IS
 5)       SELECT e, f
 6)       FROM T1
 7)       WHERE e < f
 8)       FOR UPDATE;
 9) BEGIN
10)    OPEN T1Cursor;
11)    LOOP
         /* Retrieve each row of the result of the above query
            into PL/SQL variables: */
12)        FETCH T1Cursor INTO a, b;
         /* If there are no more rows to fetch, exit the loop: */
13)        EXIT WHEN T1Cursor%NOTFOUND;
         /* Delete the current tuple: */
14)        DELETE FROM T1 WHERE CURRENT OF T1Cursor;
         /* Insert the reverse tuple: */
15)        INSERT INTO T1 VALUES(b, a);
16)    END LOOP;
      /* Free cursor used by the query. */
17)    CLOSE T1Cursor;
18) END;
19) .
20) run;
```

הדפסת משתנים

נוכל להדפיס רק משתנה שהוגדר בצורה המיוחדת VARIABLE <name> <type>, כאשר הטיפוס הוא NUMBER, CHAR(n), CHAR. נוכל להשים ערכים למשתנה זה, כאשר לפני המשתנה נשים נקודותיים. כעת, **מחוץ ל-PL/SQL** נוכל לכתוב: PRINT :<name>;

כדוגמה פשוטה ניקח את הקטע הבא:

```
VARIABLE x NUMBER

BEGIN
     :x := 1;
END;
.
run;

PRINT :x;
```

פרוצדורות

נראה דוגמה לפרוצדורה addtuple1 שמקבלת את i ומכניסה את השורה (i,'xxx') לטבלה הבאה:

```
CREATE TABLE T2 (
   a INTEGER,
   b CHAR(10)
);
```

```
CREATE PROCEDURE addtuple1(i IN NUMBER) AS
BEGIN
   INSERT INTO T2 VALUES(i, 'xxx');
END addtuple1;
.
run;
```

ניתן ליצור פרוצדורה על ידי CREATE OR REPLACE כדי לבצע מחיקה של הפרוצדורה הקודמת בשם זה, אם הייתה קיימת כזו, ולא לקבל הודעת שגיאה.

הפרוצדורה יכולה לקבל מס' כלשהו של פרמטרים, כל אחד מלווה במצב (mode) וסוג. המצבים האפשריים הם:

- IN (לקריאה בלבד) – ערך הפרמטר כן מתעדכן בערך הארגומנט בכניסה לפרוצדורה, אך הארגומנט אינו מקבל את ערך הפרמטר ביציאה ממנה.
- OUT (לכתיבה בלבד) - ערך הפרמטר לא מתעדכן בערך הארגומנט בכניסה לפרוצדורה, אך הארגומנט כן מקבל את ערך הפרמטר ביציאה ממנה.
- INOUT (לקריאה וכתיבה) - ערך הפרמטר מתעדכן בערך הארגומנט בכניסה לפרוצדורה, והארגומנט מקבל את ערך הפרמטר ביציאה ממנה.

הערה חשובה: אין לתת גודל לפרמטר מסוג CHAR או VARCHAR. הגודל ייקבע בהתאם לארגומנט שהועבר בקריאה לפרוצדורה.

אגב, אין חובה לציין את שם הפרוצדורה הנסגרת לאחר ה-END. די לכתוב END;.

הגדרת משתנים לוקאליים של הפרוצדורה **לא** תתבצע בעזרת DECLARE, אלא בצורה הבאה:

```
CREATE PROCEDURE <procedure_name> (. . . . . .)  AS
<local_var_declarations>
BEGIN
   <procedure_body>
END;
.
run;
```

השורה בסוף run; מסיימת את יצירת הפרוצדורה. היא אינה מריצה אותה. כדי להריץ את הפרוצדורה נשתמש בתוכנית PL/SQL:

```
BEGIN
   addtuple1(99);
END;
.
run;
```

הפרוצדורה הבאה גם מוסיפה שורה ל-T2, אך מקבלת את שני השדות:

```
CREATE PROCEDURE addtuple2(
   x T2.a%TYPE,
   y T2.b%TYPE)
AS
BEGIN
   INSERT INTO T2(a, b)
   VALUES(x, y);
END addtuple2;
.
run;
```

נוסיף שורה (10, 'abc') ל-T2:

```
BEGIN
   addtuple2(10, 'abc');
END;
.
run;
```

נראה דוגמה לשימוש ב-OUT:

```
CREATE TABLE T3 (
   a INTEGER,
   b INTEGER
);

CREATE PROCEDURE addtuple3(a NUMBER, b OUT NUMBER)
AS
BEGIN
   b := 4;
   INSERT INTO T3 VALUES(a, b);
END;
.
run;

DECLARE
   v NUMBER;
BEGIN
   addtuple3(10, v);
END;
.
run;
```

נשים לב, כי השמה לפרמטר שהוגדר כ-INOUT/OUT תגרום לארגומנט המתאים להתעדכן בערכו ביציאה מהפרוצדורה. לכן, לעתים ארגומנט עבור פרמטר מסוג OUT/INOUT לא יאותחל, כפי שקורה עם v בתוכנית. ברור כי ארגומנט קבוע לא יכול להיות מועבר לפרמטר OUT/INOUT.

ניתן ליצור פונקציות במקום פרוצדורות. בהגדרת פונקציה, לאחר רשימת הפרמטרים, נכתוב את המלה RETURN ולאחריה הטיפוס המוחזר:

```
CREATE FUNCTION <func_name>(<param_list>) RETURN <return_type> AS ...
```

בגוף הפונקציה נכתוב `RETURN <expression>;` כדי לצאת מהפונקציה ולהחזיר את הערך של `<expression>`.

כדי למחוק פרוצדורה או פונקציה קיימים, נכתוב:

```
drop procedure <procedure_name>;
drop function <function_name>;
```

Triggers

הדקים הם מבנה ב-PL/SQL הדומה לפרוצדורות. אך פרוצדורה נקראת ישירות דרך קריאה פרוצדוראלית,בעוד שהדק מורץ באופן עקיף ברגע שאירוע ההדק קורה. אירוע ההדק יכול להיות פקודת INSERT, DELETE או UPDATE. התזמון יכול להיות BEFORE או AFTER הפקודה. ההדק יכול להיות ברמת שורה או ברמת משפט. הדק ברמת שורה פועל **פעם אחת מיוחדת** עבור כל שורה שהושפעה מאירוע ההדק. הדק ברמת משפט פועל פעם אחת עבור כל אירוע ההדק.

זהו התחביר ליצירת הדק:

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
   {BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON <table_name>
   [FOR EACH ROW [WHEN (<trigger_condition>)]]
   <trigger_body>
```

ניתן לציין עד שלושה אירועי הדק בעזרת המלה OR. לגבי UPDATE, ניתן לכתוב UPDATE OF ורשימת attribute(s) מ-<Table_name>. במקרה זה האירוע הוא רק בעדכון שדות אלה. דוגמאות:

```
... INSERT ON R ...
... INSERT OR DELETE OR UPDATE ON R ...
... UPDATE OF A, B OR INSERT ON R ...
```

אם אנו כותבים את אופציית FOR EACH ROW, אז ההדק הוא ברמת שורה. אחרת הוא ברמת המשפט. עבור הדק ברמת השורה ניתן להגביל את ההדק בעזרת תנאי SQL שנציין ב- WHEN (ללא SubQueries). התנאי צריך להתקיים כדי שההדק יופעל. אם אנו לא כותבים את חלק ה-WHEN, ההדק מופעל בכל אירוע הדק שקורה.

<trigger_body> הוא בלוק של PL/SQL ולא רצף משפטי SQL.

עלינו לדאוג לא להיכנס למצבים בעייתיים של הדקים התלויים זה בזה, או של אילוצים התלויים בהדקים.

נראה דוגמה להדק אשר לכשנכניס שורה ל-T4 בה ערך השדה הראשון קטן מ-10, ההדק יכניס את השורה ההפוכה ל-T5:

```
CREATE TABLE T4 (a INTEGER, b CHAR(10));
CREATE TABLE T5 (c CHAR(10), d INTEGER);

CREATE TRIGGER trig1
   AFTER INSERT ON T4
   FOR EACH ROW
   WHEN (NEW.a <= 10)
   BEGIN
      INSERT INTO T5 VALUES(:NEW.b, :NEW.a);
   END trig1;
.
run;
```

המשתנים המיוחדים NEW ו-OLD משמשים להתייחסות לשורה החדשה והישנה , בהתאמה. **שימו לב**: בגוף ההדק נשים נקודותיים לפני ה- NEW וה-OLD, בעוד שב-WHEN לא נשים נקודותיים!!!

נשים לב כי מה שכתבתנו, לא מריץ את ההדק, אלא רק יוצר אותו. רק אירוע הדק – הכנסת שורה ל-T4 – תפעיל את ההדק.

למחיקת הדק נכתוב:

```
drop trigger <trigger_name>;
```

כדי לאפשר או להקפיא הדק נכתוב:

```
alter trigger <trigger_name> {disable|enable};
```

גילוי שגיאות

PL/SQL לא תמיד אומר מהם טעויות הקומפילציה. כדי לראות אותן נכתוב:

```
show errors procedure <procedure_name>;
show errors trigger <trigger_name>;
```

כדי לראות את טעות הקומפילציה האחרונה נכתוב באופן סתמי SHO ERR;