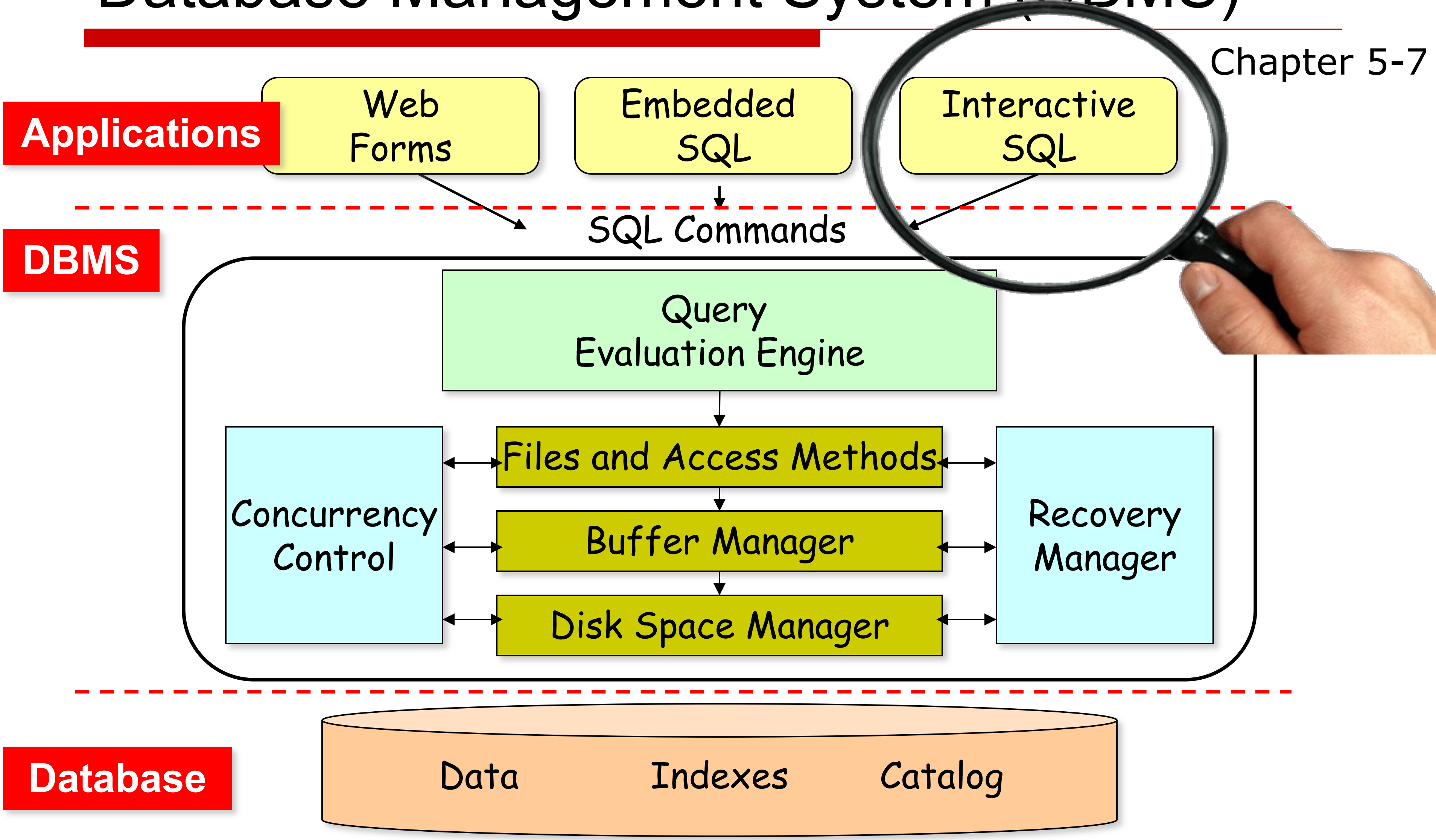
Database Management System (DBMS)

Chapter 5-7



Applications

Web Forms

Embedded SQL

Interactive SQL

SQL Commands

DBMS

Query Evaluation Engine

Concurrency Control

Files and Access Methods

Buffer Manager

Disk Space Manager

Recovery Manager

Database

Data

Indexes

Catalog

Relational Database Management Systems

- Data Abstraction
  - Overview
  - SQL Queries
  - Views
  - Integrity Constraints
  - Complex Integrity Constraints

Complex Integrity Constraints

- A constraint is expressed as a **Predicate**
  - A condition similar to the one in the `WHERE`-clause of an SQL query

- Three DDL constructs
  - Checks
  - Assertions
  - Triggers

Queries and Transactions

- Queries: requests to the DBMS to retrieve data from the database
- Updates: requests to the DMBS to insert, delete or modify existing data
- Transactions: logical grouping of query and update requests to perform a task
  - Logical unit of work (like a function/subroutine)

ACID Properties

A tomicity

C onsistency

I solation

D urability

ACID Properties

- **Atomicity**

  Either all the operations associated with a transaction happen or none of them happens

- **Consistency**

  It satisfies the integrity constraints on the database at the transaction's boundaries

- **Isolation**

  The result of the execution of concurrent transactions is the same as if transactions were executed serially

- **Durability**

  The effects of completed transactions become permanent surviving any subsequent failures

SQL Transactions

- Basic transaction statements:
  - SET TRANSACTION READ WRITE NAME <name>;
    (DECLARE TRANSACTION READ WRITE;)
  - SET TRANSACTION READ ONLY NAME <name>;
    (DECLARE TRANSACTION READ ONLY;)
  - COMMIT ;
  - ROLLBACK;

Transaction Consistency

T$_1$: `UPDATE` Accounts `SET` balance= balance - 100 `WHERE` client=7

- □ **Consistency:** It satisfies the integrity constraints on the database at the transaction's boundaries
  - □ E.g., balance is not allowed to be negative
- □ Mechanism: **Integrity Constraints (ICs)**
  - □ *Checks, Assertions, Triggers, etc.*

Transaction Atomicity

- What do we expect with Atomicity?
  - "All or nothing"

- Consider a transaction:

```
set transaction read write name 'test';
  insert into Student values (23, 'John', 'CS');
  insert into Dept values ('CS', 'ITEE');
Commit;
```

- What happens if the first insert fails, e.g., due to a referential constraint violation?
  - Is the new tuple inserted into Department? No?

Modes of Constraints Enforcement

- **NON DEFERRABLE** or **IMMEDIATE**
  - Evaluation is performed at input time
  - By default constraints are created as NON DEFERRABLE
  - It *cannot* be changed during execution
- **DEFERRED**
  - Constraints are not evaluated until commit time
- **DEFERRABLE**
  - It can be changed within a transaction to be DEFERRED using SET CONSTRAINT
- Modes can be specified when a table is created
  - INITIALLY IMMEDIATE: constraint validation happen immediately
  - INITIALLY DEFERRED: constraint validation defer until commit

Specifying Initial Evaluation Mode in Tables

```
CREATE TABLE SECTION
( SectNo sectno_dom,
  Name section_dom,
  HeadSSN ssn_dom,
  Budget budget_dom,


);
```

Specifying Initial Evaluation Mode in Tables

```
CREATE TABLE SECTION
( SectNo sectno_dom,
  Name section_dom,
  HeadSSN ssn_dom,
  Budget budget_dom,
 CONSTRAINT section_PK
   PRIMARY KEY (SectNo) DEFERRABLE,
 CONSTRAINT section_FK
   FOREIGN KEY (HeadSSN) REFERENCES LIBRARIAN(SSN)
   INITIALLY DEFERRED DEFERRABLE,
 CONSTRAINT section_name_UN UNIQUE (Name)
   DEFERRABLE INITIALLY IMMEDIATE
);
```

Changing Constraint Evaluation Mode

- It is permitted only for deferrable constraints

- Setting the constraint validation mode within a transaction
  - Set mode of all deferrable constraints

    ```
    SET CONSTRAINT ALL IMMEDIATE;
    SET CONSTRAINT ALL DEFERRED;
    ```

  - Set mode of specific deferrable constraints (list)

    ```
    SET CONSTRAINT section_budget_IC IMMEDIATE;
    SET CONSTRAINT section_budget_IC DEFERRED;
    ```

Specifying Transaction Atomicity 1

- Errors at commit time: only when **deferred constraints** are violated
  - Constraints can be deferred if specified as **deferrable** in the table schema, and
  - Deferred in the scope of the transaction
- E.g. 1, *assume the constraints are deferrable*

```
set transaction read write name ’test’;
set constraints all deferred;
insert into Student values (23, 'John', 'CS');
insert into Dept values ('CS', 'ITEE');
Commit;
```

- No constraint violation of the first insert is detected at *commit time* → the whole transaction is committed

Specifying Transaction Atomicity 2

- Errors at commit time: only when **deferred constraints** are violated
  - E.g. 2, *assume the constraints are deferrable and assume SID 23 exists in that Database*

```
set transaction read write name ’test’;
set constraints all deferred;
insert into Student values (23, 'John', 'CS');
insert into Dept values ('CS', 'ITEE');
Commit;
```

- The constraint violation of the first insert is detected at *commit time* → the whole transaction is rollback

Specifying Transaction Atomicity 3

- Errors at commit time: only when **deferred constraints** are violated
  - E.g. 3, *assume the primary key constraints are non-deferrable but the foreign key constraints are deferrable and assume SID 23 exists in that Database*

```
set transaction read write name ’test’;
set constraints all deferred;
insert into Student values (23, 'John', 'CS');
insert into Dept values ('CS', 'ITEE');
Commit;
```

- What would happen?

Complex Integrity Constraints

- Three DDL constructs
  - Checks
  - Assertions
  - Triggers

Example Schema

SID	*Name*	*Age*	*GPA*	*Major*
546007	Peter	18	3.8	IT
546100	Bob	19	6.65	Cinema
546500	Peter	20	4.7	History

Example Schema

```
CREATE TABLE Student (
    Sid INTEGER,
    Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major CHAR(10),

    PRIMARY KEY (Sid));
```

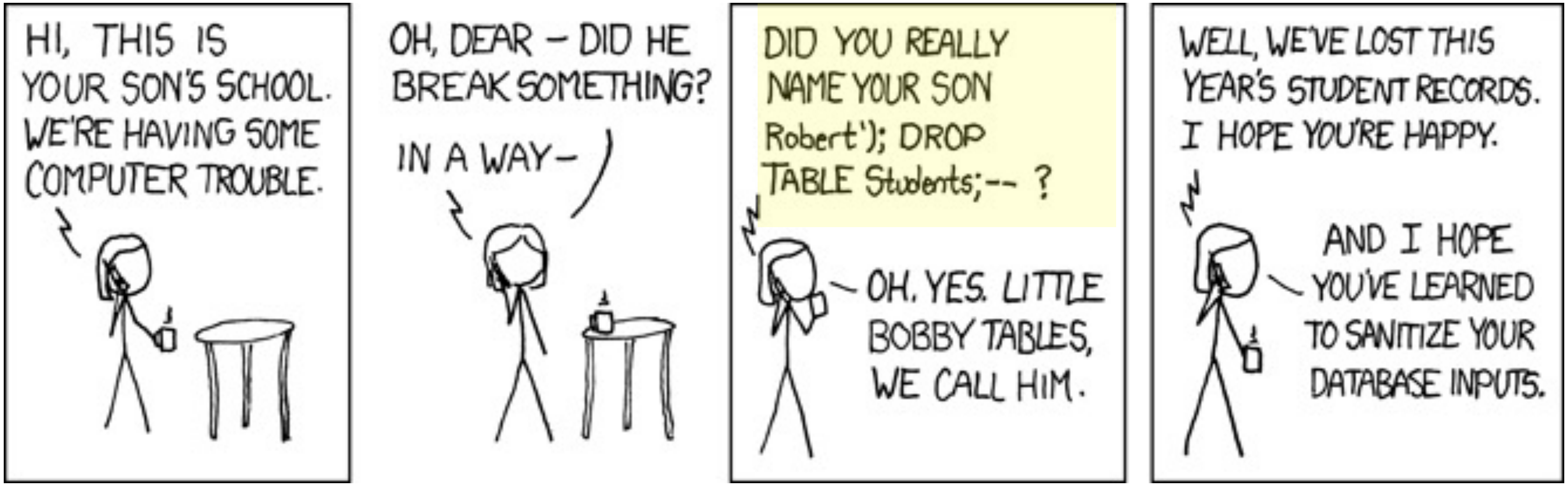
CHECK Constraint 1

```
CREATE TABLE Student (
    Sid INTEGER,
    Name CHAR(20),
    Age INTEGER,
    GPA REAL,
        CHECK (GPA>=0.0 AND GPA <= 7.0);
    Major CHAR(10),

    PRIMARY KEY (Sid));
```

CHECK Constraint 2

```
CREATE TABLE Student (
    Sid INTEGER,
    Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major CHAR(10),
        CHECK (Major IN (‘IT’, ‘Cinema’, ‘History’));

    PRIMARY KEY (Sid));
```

Be Careful with Your Database Inputs ☺


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

CHECK Constraint and DOMAIN

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN (‘IT’, ‘Cinema’, ‘History’));


CREATE TABLE Student (
    Sid INTEGER,
    Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    PRIMARY KEY (Sid));
```

CHECK: Attribute- vs. Tuple-based

- CHECK prohibits an operation on a table that would violate a constraint
- CHECK clause restricts acceptable attribute values according to some definition
  - **Attribute-based**
- CHECK is also used as a **tuple-based** constraint:
  - Apply to each tuple individually
  - Checked whenever a tuple is inserted or modified
  - See next example…

Example

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN (‘IT’ ,‘Cinema’, ‘History’));
CREATE TABLE Student (
    Sid INTEGER,
    Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    Minor …, what constraints are needed for Minor?
    PRIMARY KEY (Sid));
```

IC1: Minor IN …

IC2: Minor ≠ Major

Example: Attribute-based

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN (‘IT’, ‘Cinema’, ‘History’));
CREATE TABLE Student (
    Sid INTEGER,
    Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    Minor M_Code,
    PRIMARY KEY (Sid));
```

IC1: attribute-based

Example: Attribute- and Tuple-based

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN ('IT', 'Cinema', 'History'));
CREATE TABLE Student (
    Sid INTEGER,
    Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    Minor M_Code, CHECK (Major != Minor);
    PRIMARY KEY (Sid));
```

IC1: attribute-based

IC2: tuple-based

Naming Constraints

- A constraint may be given a name using the keyword **`CONSTRAINT`**
  - E.g., **`CONSTRAINT`** Major_Minor

- Advantages of naming a constraint:
  - Facilitates editing
  - Identifies a particular constraint
    1. For reporting
    2. For constraint management

Naming Constraints

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN (‘IT’, ‘Cinema’, ‘History’));
CREATE TABLE Student (
    Sid INTEGER,
    Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    Minor M_Code,
    CONSTRAINT Major_Minor
        CHECK (Major != Minor););
```

Constraint Management

```
ALTER TABLE Student DROP CONSTRAINT Major_Minor;
```

```
ALTER TABLE Student ADD CONSTRAINT Major_Minor
      CHECK (Major != Minor);
```

- To modify a constraint:
  - Drop it first and then add a new one

Assertions

- Similar to `CHECK` but they are **global** constraints

```
CREATE ASSERTION <assertion_name>
CHECK <condition>;
```

  - Global: schema-based
  - <condition> must be TRUE for each database state
- Examples:
  - # of IT stduents cannot exceed 1800
  - # of students in a prac cannot exceed lab capacity
  - …

Assertions

```
CREATE ASSERTION <assertion_name>
CHECK NOT EXISTS (vquery);
```

1. Specify a query <*vquery*> such that:
   vquery selects any tuple that **violates** *<condition>*

2. Include vquery inside a **`NOT EXISTS`** clause

Assertions

```
CREATE ASSERTION <assertion_name>
CHECK NOT EXISTS (vquery);
```

Result of *vquery*	**NOT EXISTS (*vquery*)**	**CHECK**
Empty (no tuples violate the condition)	TRUE	Satisfied
Not Empty (some tuples violate the condition)	FALSE	Violated

Example Schema

SID	*Name*	*Age*	*GPA*	*Major_Code*
546007	Peter	18	3.8	0
546100	Bob	19	3.65	50
546500	Peter	20	3.7	1

Major_Code	*Major_name*
0	IT
1	History
…	
50	Cinema

Example

- Number of students in any major cannot exceed 1800

```
CREATE ASSERTION Major_Limit
CHECK  NOT EXISTS (
                 SELECT Major_Code, COUNT(*)
                 FROM Student
                 GROUP BY Major_Code
                 HAVING COUNT(*) > 1800 );
```

Example

- The Number of students cannot exceed 1800 in each of the IT or Cinema majors

```
CREATE ASSERTION Major_Limit
CHECK NOT EXISTS (
      SELECT Major_Code, COUNT(*)
      FROM Student AS S , Major AS M
      WHERE S.major_code = M.major_code
      AND (M.major_name=“IT” OR M.major_name=“Cinema”)
      GROUP BY Major_Code
      HAVING COUNT(*) > 1800 );
```

Triggers

- A trigger consists of 3 parts:
  1. Event(s),
  2. Condition, and
  3. Action

  - E.g., notify the Dean whenever the number of students in any major exceeds 1800

Triggers vs. Assertions

- Assertion
  - Condition must be true for each database state
  - DBMS rejects operations that violate such condition
- Trigger
  - DBMS takes a certain **action** when condition is true
  - Action could be: stored procedure, SQL statements, Rollback, etc.

Example

- Notify the Dean when the # of students in any major exceeds 1800

`CREATE TRIGGER` Major_Limit

Event(s)

Condition

Action

Example

- Notify the Dean when the # of students in any major exceeds 1800

CREATE TRIGGER Major_Limit

Event(s)

```
WHEN( EXISTS (
                 SELECT Major_Code, COUNT(*)
                 FROM Student
                 GROUP BY Major_Code
                 HAVING COUNT(*)  > 1800 ))
```

Action

Example

- Notify the Dean when the # of students in any major exceeds 1800

```
CREATE TRIGGER Major_Limit
```

Event(s)

```
WHEN( EXISTS (
                SELECT Major_Code, COUNT(*)
                FROM Student
                GROUP BY Major_Code
                HAVING COUNT(*)  > 1800 ))
```

```
CALL email_dean(Major_code);
```

Example

- Notify the Dean when the # of students in any major exceeds 1800

```
CREATE TRIGGER Major_Limit
```

```
AFTER INSERT OR UPDATE OF Major_Code
ON Student
```

```
WHEN( EXISTS (
                 SELECT Major_Code, COUNT(*)
                 FROM Student
                 GROUP BY Major_Code
                 HAVING COUNT(*)  > 1800 ))
```

```
CALL email_dean(Major_code);
```

Triggers (SQL99)

```
CREATE or REPLACE TRIGGER <trigger-name>
     <time events> ON <list-of-tables>
     REFERENCING { NEW | OLD} AS  <user-name>
  [FOR EACH { ROW | STATEMENT} ]
    [WHEN (<Predicate>)]
  <action>
```

- time: **before** or **after**
- events: **Insert, Delete, Update [of <list of attributes>]**
- **NEW & OLD** refer to new & old (existing) tuples/table respectively
- The REFERENCING clause assigns aliases to NEW and OLD
- action: Stored procedure or

  `BEGIN ATOMIC {<SQL procedural statements>} END`

Oracle Example: Statement Trigger

- Statement-level trigger fires **once** by the triggering statement

- No WHEN-clause in the definition of statement trigger

-
```
CREATE OR REPLACE TRIGGER Audit_Updater
   AFTER
   INSERT OR DELETE OR UPDATE
   ON STUDENTS
BEGIN
     INSERT INTO AUDIT_Table VALUES (`STUDENT’, sysdate);
END;
/
```

- The end slash (“/”) installs and activates the trigger

Oracle Example: Row-Level Trigger

- Row- or tuple-level trigger fires once **for each row** affected by the triggering statement

-
  ```
  CREATE OR REPLACE TRIGGER trigger_deans_list
     AFTER INSERT ON STUDENTS
     REFERENCING NEW AS newRow
  FOR EACH ROW
      WHEN (newRow.GPA > 6.0)
  BEGIN
       INSERT INTO DL VALUES ( :newRow.SID, :newRow.GPA );
  END;
  /
  ```

- Scope Rules: In the trigger body, NEW and OLD must be preceded by a colon (":"), but in the WHEN clause (triggering condition), they do not have a preceding colon!