



UNIVERSITY *of* WASHINGTON

Week 8: The Rest of SQL



AGENDA

> Finalize SQL Concepts

- Constraints
- Stored Procedures
- Triggers
- Views
- Indexes



UNIVERSITY *of* WASHINGTON

Constraints Cont.



Domain Constraints Review

- > **Determines the allowed value of a field:**
 - Value must be greater than this or that
 - Must be in a predefined list
 - Must be found in a specified column (Foreign Keys)



Foreign Key Constraint Syntax

```
CREATE TABLE Employee (  
    EmployeeId INT PRIMARY KEY AUTO_INCREMENT  
    , FirstName VARCHAR(50) NOT NULL  
    , LastName VARCHAR(50) NOT NULL  
    , ManagerId INT NULL  
    , FOREIGN KEY fk_mgrid (ManagerId)  
        REFERENCES Employee (EmployeeId)  
        ON DELETE RESTRICT  
        ON UPDATE CASCADE  
);
```



Constraints: Checks (SQL-99 Standard)

Enforces expected values are stored in a tuple on insert (AND UPDATE) to ensure data validity

Broader scope than Domain/Referential Integrity Checks

Can apply to a single attribute or the entire tuple



Constraints: Checks (SQL-99 Standard)

```
CREATE TABLE my_table (  
    big_number int CHECK (big_number >= 100000000)  
    , course_status varchar(32)  
);
```

```
ALTER TABLE my_table  
ADD CHECK (course_status IN ('Completed','Registered'));
```



Constraints: Checks (SQL-99 Standard)

```
CREATE TABLE my_table (  
    gender char(1) CHECK (gender IN ('F','M'))  
    , is_pregnant boolean  
    , CHECK (!(gender = 'm' and is_pregnant = 1))  
);
```

If the condition is false for that tuple, then the constraint is violated and the insertion or update statement that caused the violation is rejected.



Assertions (also not in MariaDB)

- > A boolean-valued SQL expression that must be true at all times.
- > Not supported in many DBMSs even though it is part of SQL standards

```
create assertion
AT_MOST_ONE_PRESIDENT as CHECK
((select count(*)
  from EMP e
 where e.JOB = 'PRESIDENT') <= 1
)
```

```
create assertion
NO_TRAINERS_IN_BOSTON as CHECK
(not exists
  (select 'trainer in Boston'
   from EMP e, DEPT d
  where e.DEPTNO = d.DEPTNO
        and e.JOB  = 'TRAINER'
        and d.LOC   = 'BOSTON')
)
```





UNIVERSITY *of* WASHINGTON

STORED PROCEDURES



Stored Procedures

- A stored procedure is an executable database object that contains SQL statements.
- Stored procedures are *precompiled*. That means that the *execution plan* for the SQL code is compiled the first time the procedure is executed and is then saved in its compiled form.
- Because it's precompiled, a stored procedure executes faster than an equivalent SQL script.
- You use the EXEC statement to run, or *call* a procedure. If this statement is the first line in a batch, the EXEC keyword is optional but is often coded for clarity.
- You can call a stored procedure from within another stored procedure.
- Can restrict and control access to a database.

PROCEDURE Syntax

```
CREATE PROCEDURE <name> (<parameters>)  
<local declarations>  
<procedure body> ;
```

```
ALTER PROCEDURE <name> (<parameters>)  
<local declarations>  
<procedure body> ;
```

```
DROP PROCEDURE <name>;
```

Procedure Example

```
DELIMITER $  
CREATE PROCEDURE spGetMovies()  
BEGIN  
    SELECT * FROM Movies;  
END $  
DELIMITER ;
```

```
CALL spGetMovies();
```

```
drop procedure `moviedb`.`spGetMovies`;
```

Parameters

- `CREATE PROCEDURE proc()` //Parameter list is empty
- `CREATE PROCEDURE proc(IN varname DATA-TYPE)` //One input parameter. The word IN is optional because parameters are IN (input) by default.
- `CREATE PROCEDURE proc(OUT varname DATA-TYPE)` //One output parameter.
- `CREATE PROCEDURE proc(INOUT varname DATA-TYPE)` //One parameter which is both input and output.

Input parameter

```
DELIMITER $  
CREATE PROCEDURE spGetMovieByTitle(a_title VARCHAR(100))  
BEGIN  
    SELECT * FROM Movies WHERE title = a_title;  
END $  
DELIMITER ;  
  
CALL spGetMovieByTitle('Star Wars')
```

Output parameter

```
DELIMITER $  
CREATE PROCEDURE spGetMovieCount(OUT count int(11))  
BEGIN  
    SELECT COUNT(*) INTO count FROM Movies;  
END $  
DELIMITER ;  
  
CALL spGetMovieCount(@count);  
SELECT @count;
```


INOUT parameter

```
DROP PROCEDURE `moviedb`.`spGetMovieLengthByYear`;
DELIMITER $
CREATE PROCEDURE spGetMovieLengthByYear(INOUT lengthOrYear int(11))
BEGIN
    SELECT SUM(Length) INTO lengthOrYear
    FROM Movies
    WHERE year = lengthOrYear;
END $
DELIMITER ;
```

```
SET @count = 2013;
CALL spGetMovieLengthByYear(@count);
SELECT @count;
```

Declaring variables

```
DROP PROCEDURE `moviedb`.`spFindStudioAddress`;
DELIMITER $
CREATE PROCEDURE spFindStudioAddress(IN my_title VARCHAR(100), IN my_year INT)
BEGIN
    DECLARE cert INT;
    SELECT `producerC#` INTO cert
        FROM Movies
        WHERE title = my_title AND year = my_year;

    SELECT address
        FROM Studio
        WHERE `presC#` = cert;
END $
DELIMITER ;

CALL spFindStudioAddress('Star Wars', 1977);
```

CONTROL STRUCTURES

```
IF condition THEN
    statements
ELSE
    statements
END IF;
```

Use ELSEIF for mutually exclusive conditions.

```
DROP PROCEDURE `moviedb`.`spGetLength`;
DELIMITER $
CREATE PROCEDURE spGetLength(IN my_year INT, OUT totalLength INT)
BEGIN
    SELECT SUM(length) INTO totalLength FROM Movies WHERE year = my_year;
    IF totalLength IS NULL THEN
        SET totalLength = 0;
    END IF;
END $

DELIMITER ;

CALL spGetLength(2001, @length);
SELECT @length;
```



Loops

```
WHILE <condition> DO  
    statements  
END WHILE;
```

```
Label: LOOP  
    statements  
END LOOP label;
```

Error Handling: SQLSTATE

- A special variable, called SQLSTATE in the SQL standard , serves to connect the host language program with the SQL execution system.
- The type of SQLSTATE is an array of five characters. Each time a function of the SQL library is called, a code is put in the variable SQLSTATE that indicates any problems found during that call.
- The SQL standard also specifies a large number of five character codes and their meanings . For example , ' 00000 ' indicates that no error condition occurred, and '02000 ' indicates that a tuple requested as part of the answer to a SQL query could not be found.

<https://mariadb.com/kb/en/library/mariadb-error-codes/>

Syntax to declare a condition for a SQLState:

DECLARE < name> CONDITION FOR SQLSTATE < value>;

Exceptions

- An Exception handler can be invoked whenever one of a list of the SQL error codes appears in SQLSTATE during the execution of a statement or list of statements.
- Each exception handler is associated with a block of code, delineated by BEGIN . . . END.
- The handler appears within this block, and it applies only to statements within the block.
- The components of the handler are:
 - A list of exception conditions that invoke the handler when raised.
 - Code to be executed when one of the associated exceptions is raised.
 - An indication of where to go after the handler has finished its work.

*DECLARE <where to go next> HANDLER FOR <condition list>
<statement>*

<where to go> can be

- CONTINUE , which means that after executing the statement in the handler declaration , we execute the statement after the one that raised the exception.
- EXIT , which means that after executing the handler's statement , control leaves the BEGIN . . . END block in which the handler is declared . The statement after this block is executed next .
- UNDO , which is the same as EXIT , except that any changes to the database or local variables that were made by the statements of the block executed so far are undone . That is , the block is a transaction , which is aborted by the exception .

Cursors

- A way to iterate through some rows of data returned from one query.

DECLARE <Cursor> CURSOR FOR ..

OPEN <Cursor>

FETCH FROM <Cursor> into ..

CLOSE <Cursor>

Cursors Example

```
DELIMITER $
CREATE PROCEDURE MeanVar (
    IN s CHAR(15),
    OUT mean REAL,
    OUT variance REAL)
BEGIN
    DECLARE newLength INTEGER ;
    DECLARE movieCount INTEGER ;

    DECLARE done INT DEFAULT FALSE;
    DECLARE MovieCursor CURSOR FOR SELECT length FROM Movies WHERE studioName = s ;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    SET mean = 0.0;
    SET variance = 0.0;
    SET movieCount = 0;

    OPEN MovieCursor;
    movieLoop: LOOP
        FETCH FROM MovieCursor INTO newLength;
        IF done THEN LEAVE movieLoop;
        END IF;
        SET movieCount = movieCount + 1 ;
        SET mean = mean + newLength ;
        SET variance = variance + newLength * newLength;
    END LOOP movieLoop;
    SET mean = mean / movieCount ;
    SET variance = variance / movieCount - mean * mean;
    CLOSE MovieCursor ;
END $
DELIMITER ;

CALL MeanVar('Fox', @mean, @variance);
SELECT @mean, @variance;
```


Functions

- Functions must return a value and can be used in SQL queries.

CREATE FUNCTION <name> (<parameters>)

RETURNS <type>

 <local declarations>

<function body> ;

Function Example

```
DELIMITER $
CREATE FUNCTION GetYear ( t VARCHAR(255) )
    RETURNS INTEGER
BEGIN
    DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
    DECLARE Too_Many CONDITION FOR SQLSTATE '21000' ;
    DECLARE EXIT HANDLER FOR Not_Found, Too_Many RETURN NULL ;

    RETURN ( SELECT year FROM Movies WHERE title = t ) ;
END $
DELIMITER ;

SELECT GetYear('Captain Phillips');
SELECT GetYear('Captain Morgan');
SELECT GetYear('Star Wars');

INSERT INTO StarsIn ( movieTitle , movieYear , starName ) VALUES ( 'Remember the Titans' ,
    GetYear('Remember the Titans' ) , 'Denzel Washington' ) ;
```

© CourseSmart

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

write PSM procedures or functions to perform the following tasks:

- a) Given the name of a movie studio, produce the net worth of its president.
- b) Given a name and address, return 1 if the person is a movie star but not an executive, 2 if the person is an executive but not a star, 3 if both, and 4 if neither.
- ! c) Given a studio name, assign to output parameters the titles of the two longest movies by that studio. Assign NULL to one or both parameters if there is no such movie (e.g., if there is only one movie by a studio, there is no “second-longest”).
- ! d) Given a star name, find the earliest (lowest year) movie of more than 120 minutes length in which they appeared. If there is no such movie, return the year 0.
- e) Given an address, find the name of the unique star with that address if there is exactly one, and return NULL if there is none or more than one.
- f) Given the name of a star, delete them from **MovieStar** and delete all their movies from **StarsIn** and **Movies**.

GIVEN NAME, GIVE PREZ NETWORTH

```
DELIMITER $  
CREATE PROCEDURE getPrezWorth(studio_name VARCHAR(100)  
                             , OUT worth int)  
BEGIN  
    SELECT ME.netWorth into worth  
    FROM Studio S  
    JOIN MovieExec ME  
        ON ME.cert# = S.presC#  
    WHERE S.name = studio_name;  
END $  
DELIMITER ;  
  
CALL getPrezWorth('Fox',@var1);  
SELECT @var1;
```


W

Name+Address = What kind of person?

```
DELIMITER $
CREATE FUNCTION getPersonToken(person VARCHAR(50)
                                , address VARCHAR(50))
RETURNS INTEGER -- 1 STAR ONLY, 2 EXEC ONLY, 3 BOTH, 4 NEITHER
DECLARE isStar INT; DECLARE isExec INT;
BEGIN
    SELECT COUNT(*) INTO isStar FROM MovieStar MS    -- only works because name is a key
    WHERE MS.name = person and MS.address = address;
    SELECT COUNT(*) INTO isExec FROM MovieExec ME    -- only works because name is a key
    WHERE ME.name = person and ME.address = address;
    IF isStar+isExec = 0 THEN RETURN(4)
    ELSE RETURN (isStar + 2*isExec)
    END IF;
END $
DELIMITER ;

SELECT getPersonToken('Mark Hamill','124 Place');
```





UNIVERSITY *of* WASHINGTON

TRIGGERS



W

Triggers

- > Also called Event Condition Action rules or ECA rules.
- > Awakened when certain events specified occur – inserts, deletes or updates to a particular relation.
- > Once awakened, trigger tests a condition.
- > If condition is satisfied, the action associated with the trigger is performed by the DBMS.



Trigger Example

```
1) CREATE TRIGGER NetWorthTrigger
2) AFTER UPDATE OF netWorth ON MovieExec
3) REFERENCING
4)     OLD ROW AS OldTuple,
5)     NEW ROW AS NewTuple
6) FOR EACH ROW
7) WHEN (OldTuple.netWorth > NewTuple.netWorth)
8)     UPDATE MovieExec
9)     SET netWorth = OldTuple.netWorth
10)    WHERE cert# = NewTuple.cert#;
```



A Maria DB Working Example

```
delimiter //
CREATE TRIGGER StudioTrigger
AFTER INSERT ON Movies FOR EACH ROW
BEGIN
    SELECT name INTO @name FROM studio WHERE name = NEW.studioname;
    IF @name IS NULL THEN
        INSERT INTO studio
            VALUES(NEW.studioname, NULL, NEW.`producerc#`);
    END IF;
END;
//

delimiter ;
```



Trigger Options

- > Perform some action: AFTER or BEFORE
- > One of these events: UPDATE, INSERT, DELETE
- > WHEN a condition holds (optional)
- > “Some action” can be multiple statements (wrapped in BEGIN and END)
- > Run upon each event, or for each statement.



Trigger Example By Statement

Keeping the total netWorth of executives over \$500,000.

This statement essentially rolls back the update...

Why do we have to do this “AFTER UPDATE” and not before?

```
1) CREATE TRIGGER AvgNetWorthTrigger
2) AFTER UPDATE OF netWorth ON MovieExec
3) REFERENCING
4)     OLD TABLE AS OldStuff,
5)     NEW TABLE AS NewStuff
6) FOR EACH STATEMENT
7) WHEN (500000 > (SELECT AVG(netWorth) FROM MovieExec))
8) BEGIN
9)     DELETE FROM MovieExec
10)    WHERE (name, address, cert#, netWorth) IN NewStuff;
11)    INSERT INTO MovieExec
12)        (SELECT * FROM OldStuff);
13) END;
```



Another MariaDB Example

```
delimiter //  
CREATE TRIGGER ModelExistsTrigger  
AFTER INSERT ON Printer FOR EACH ROW  
BEGIN  
    SELECT MODEL INTO @model FROM Product WHERE model = NEW.model;  
    IF @model IS NULL THEN  
        INSERT INTO Product VALUES('Unknown', NEW.model, 'Printer');  
    END IF;  
END;
```

```
delimiter ;
```

```
INSERT INTO Printer VALUES (3008, true, 'laser-jet', 299);
```





UNIVERSITY *of* WASHINGTON

Views and Indexes



W

Views

- Views provide users controlled access to tables
- Base Table – table containing the raw data
- Dynamic View
 - A “virtual table” created dynamically upon request by a user
 - No data actually stored; instead data from base table made available to user
 - Based on SQL SELECT statement on base tables or other views
- Materialized View
 - Copy or replication of data
 - Data actually stored
 - Must be refreshed periodically to match the corresponding base tables

Creating Virtual Views

- CREATE VIEW < view-name> AS < view-definition> ;

```
CREATE VIEW ParamountMovies AS  
SELECT title, year FROM Movies  
WHERE studioName = 'Paramount';
```

```
CREATE VIEW MovieProd AS  
SELECT title, name  
FROM Movies, MovieExec  
WHERE `producerC#` = `cert#`;
```

```
CREATE VIEW MovieProd(movieTitle, prodName) AS  
SELECT title, name  
FROM Movies, MovieExec  
WHERE `producerC#` = `cert#`;
```

Querying Views

- A view may be queried exactly as if it were a stored table.

```
SELECT title FROM ParamountMovies  
WHERE year = 1970;
```

```
SELECT DISTINCT starName  
FROM ParamountMovies, StarsIn  
WHERE title = movieTitle  
AND year = movieYear;
```


Modifying Views

- Updatable views make it possible to translate the modification of the view into an equivalent modification on a base table, and the modification can be done to the base table instead.
- To drop a view, `DROP VIEW ParamountMovies;`

Updatable Views

- A view will be updatable when
 - The WHERE clause must not involve R in a subquery.
 - The FROM clause can only consist of one occurrence of R and no other relation.
 - The list in the SELECT clause must include enough attributes that for every tuple inserted into the view , we can fill the other attributes out with NULL values or the proper default.

```
INSERT INTO ParamountMovies  
VALUES('Star Trek', 1979);
```

```
DELETE FROM ParamountMovies  
WHERE title LIKE '%Trek%';
```

```
UPDATE ParamountMovies  
SET year = 1979  
WHERE title = 'Star Trek the Movie';
```

Advantages & Disadvantages of Views

Advantages

- Simplify query commands
- Assist with data security
- Enhance programming productivity
- Contain most current base table data
- Use little storage space
- Provide customized view for user
- Establish physical data independence

Disadvantages

- Use processing time each time view is referenced
- May or may not be directly updateable

Create Views Exercise

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Construct the following views:

- a) A view **RichExec** giving the name, address, certificate number and net worth of all executives with a net worth of at least \$10,000,000.
- b) A view **StudioPres** giving the name, address, and certificate number of all executives who are studio presidents.
- c) A view **ExecutiveStar** giving the name, address, gender, birth date, certificate number, and net worth of all individuals who are both executives and stars.

Exercise 8.1.2: Write each of the queries below, using one or more of the views from Exercise 8.1.1 and no base tables.

- a) Find the names of females who are both stars and executives.
- b) Find the names of those executives who are both studio presidents and worth at least \$10,000,000.

Indexes

- An index on an attribute A of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A.
- We could think of the index as a binary search tree of (key , value) pairs , in which a key a (one of the values that attribute A may have) is associated with a “value” that is the set of locations of the tuples that have a in the component for attribute A.
- B-Tree is the data structure used in the implementation of DBMS.

Motivation for Indexes

- When relations are very large, it becomes expensive to scan all the tuples of a relation to find those (perhaps very few) tuples that match a given condition.

```
SELECT *  
FROM Movies  
WHERE studioName = 'Disney' AND year = 1990;
```

- What if there were 10000 tuples of which only 200 were made in 1990? What if there were only 10 that match both conditions?
- Indexes are also useful in queries that involve a join.

```
SELECT name  
FROM Movies, MovieExec  
WHERE title = 'Star Wars' AND producerC# = cert#;
```

© Course

Creating and Dropping Indexes

- Use CREATE INDEX

```
CREATE INDEX YearIndex ON Movies(year);
```

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

- DROP INDEX YearIndex;

Selection of Indexes

- Choosing which indexes to create requires the database designer to analyze a trade-off.
- Two important factors to consider are:
 - Speeds up greatly the execution of queries and joins specified on the attribute.
 - Makes modifications more complex and time-consuming.
- To choose indexes, we need to understand where the time is spent answering a query. Data is brought in pages into memory. It costs little more time to examine all the tuples on a page than to examine only one.
- Most useful index is on the key since it will be used for joins and to retrieve a tuple and guaranteed to return only one tuple.

Create Indexes Exercise

© CourseSmart

```
Movies(title, year, length, genre, studioName, producerC#)  
StarsIn(movieTitle, movieYear, starName)  
MovieExec(name, address, cert#, netWorth)  
Studio(name, address, presC#)
```

Declare indexes on the following attributes or combination of attributes:

- a) studioName.
- b) address of MovieExec.
- c) genre and length.