# CSCI 5333 DBMS Classroom Notes

**11/9/2021**

**3. Views**

* Views are *virtual* table *derived* from other tables.
* Some advantages of using views:
	1. Better data abstraction
	2. Logical data independence
	3. Better consistency
	4. Possible centralized security control
	5. Can be more efficient
* Some disadvantages:
	1. More work.
	2. Can be inefficient.
	3. Complicated especially when views are updated.
* MySQL Create View Manual: search for "mysql view manual"

**4. Stored Procedures**

* Routines stored by the DBMS that can be called.
* Routines can be
	1. functions: return a value (but not a table): should not have side effects, or
	2. procedures must have side effects: do not return a value.
* Known as SQL/PSM (SQL/Persistent Stored Modules) in MySQL.
	1. Can execute SQL statements
	2. Include general programming constructs
* Some advantages of stored procedures
	1. Enforcing application constraints and requirements in lower levels.
	2. Providing consistency and security control.
	3. Possible performance optimization by both the developers and the DBMS.
	4. Sharing among DB applications: code reuse.
* Some disadvantages of stored procedures:
	1. Use up DB server's resources.
	2. Potentially inefficient because of limitations in language constructs.
	3. Potentially harder to develop because of relative lack of libraries and difficulty in debugging.
	4. Can be vendor specific.
* In general, stored procedures should be used more often.
* MySQL stored procedures:
	1. <http://dev.mysql.com/doc/refman/5.6/en/stored-routines.html>
	2. <http://dev.mysql.com/doc/refman/5.6/en/faqs-stored-procs.html#qandaitem-B-4-1-2>

***Example:***

DROP PROCEDURE AllSuppliers;

--  A very simple stored procedure
DELIMITER //

CREATE PROCEDURE AllSuppliers()
BEGIN
   SELECT \*  FROM Supplier;
END //

DELIMITER ;

--  Call
CALL AllSuppliers;

Note:

* Use of the DELIMITER command to redefine // as the delimiter indicating the end of the stored procedure. Otherwise, the default ; is the delimiter indicating the end of the stored procedure.
* No parameters for the procedure.
* A procedure does not return any value and accomplish its goal through *side effects*.
* The result is displayed.

***Example:***

DROP FUNCTION GetAllSuppliers;

--  A very simple stored function
DELIMITER //

CREATE FUNCTION GetAllSuppliers() RETURNS INT
BEGIN
   DECLARE count INT DEFAULT 0;
   SET count = 0;    -- Not really needed.

   SELECT COUNT(\*) INTO count FROM Supplier;

   -- The following is not allowed as result set in function.
   -- SELECT \*  FROM Supplier;

   RETURN count;
END //

DELIMITER ;

--  Call
SELECT GetAllSuppliers();

Note:

* A function returns a value and does not accomplish its goal through *side effects*.
* Note variable declaration and assignment.
* The scope of the variable is within the function.
* On the other hand, a session variable (starts with a @) has the session as its scope. Use them very carefully.

***Example:***

--  Parts supplied by a supplier

DROP PROCEDURE PartSupplied;
DELIMITER //

CREATE PROCEDURE PartSupplied(
   IN snum VARCHAR(9),
   IN max INT,                  --  Maximum number of records shown
   OUT numberPartsSupplied INT  --  Number of different parts supplied
)
BEGIN
   SELECT COUNT(\*) INTO numberPartsSupplied
   FROM Supply S
   WHERE S.SNUM = snum;

   SELECT P.PNUM, P.PNAME, S.Quantity
   FROM Supply S, Part P
   WHERE S.SNUM = snum
   AND S.PNUM = P.PNUM
   LIMIT max;

END //

DELIMITER ;

--  Call

SET @numberPartsSupplied = 0;
CALL PartSupplied('S2', 2, @numberPartsSupplied);
SELECT @numberPartsSupplied;

 Note:

* Parameter passing mode: IN, OUT and IN OUT. This is similar to the language Ada.
* Use of the parameter max in the LIMIT clause. There are limitations on where parameters can be used in SQL statements.
* Use of the session variable @numberPartsSupplied.

**Stored Program Syntax**

* Syntax for MySQL's stored programs can be found in compound statement section. Google it.
* Various high level language constructs are supported, such as variable declaration, conditional statements, control statements, etc.
* In particular, *cursor* is supported for allowing iteration through relations: <https://dev.mysql.com/doc/refman/5.7/en/cursors.html>. Look for the latest version.

**5. Triggers**

* Triggers allow *event-driven* programming.
* A trigger is activated when certain events occur. Unlike stored routines, trigger is not called directly.
* Four questions:
* (1) Events:
	+ Insert a row
	+ Update a row
	+ Delete a row
* (2) Event handlers: Actions can be executed:
	+ Before the event
	+ After the event
* (3) Event information: Information can be obtained through:
	+ old:
		- the old value of the row before the event.
		- for update and delete events
	+ new:
		- the new value of the row after the event.
		- for update and insert events
* (4) Event propagation: no
* Advantages of triggers:
	+ event driven model suit certain kind of tasks better.
	+ implementation of consistency check.
	+ implementation of business logic and integrity check.
* Disadvantages:
	+ Relatively invisible and possibly overlook by developers.
	+ Potential performance issues.
	+ Potential complicated interactions.

***Example:***

Using the Sakila database, consider the tables inventory and rental.

inventory (inventory\_id, film\_id, store\_id, last\_update)
rental (rental\_id, rental-date, inventory\_id, customer\_id, return\_date, staff\_id, last\_update)

The stored function inventory\_in\_stock(p\_inventory\_id INT) checks whether an inventory item is in stock. Its logic:

"AN ITEM IS IN-STOCK IF THERE ARE EITHER NO ROWS IN THE rental TABLE FOR THE ITEM OR ALL ROWS HAVE return\_date POPULATED."

CREATE FUNCTION inventory\_in\_stock(p\_inventory\_id INT) RETURNS BOOLEAN
READS SQL DATA
BEGIN
    DECLARE v\_rentals INT;
    DECLARE v\_out     INT;

    #AN ITEM IS IN-STOCK IF THERE ARE EITHER NO ROWS IN THE rental TABLE
    #FOR THE ITEM OR ALL ROWS HAVE return\_date POPULATED

   -- Redundant as the test is covered by the next SQL statement
    SELECT COUNT(\*) INTO v\_rentals
    FROM rental
    WHERE inventory\_id = p\_inventory\_id;

    IF v\_rentals = 0 THEN
      RETURN TRUE;
    END IF;

    SELECT COUNT(rental\_id) INTO v\_out
    FROM inventory LEFT JOIN rental USING(inventory\_id)
    WHERE inventory.inventory\_id = p\_inventory\_id
    AND rental.return\_date IS NULL;

    IF v\_out > 0 THEN
      RETURN FALSE;
    ELSE
      RETURN TRUE;
    END IF;
END $$

DELIMITER ;

This can be inefficient.

We consider modifying the table inventory to speed up performance.

inventory (inventory\_id, film\_id, store\_id, lost, in\_stock, last\_update)

Two columns are added:

* lost: whether the inventory item has been lost; default FALSE.
* in\_stock: whether the inventory item is in stock; default TRUE.

We may consider that the column in\_stock as a *derived* column. Its value can be derived from other columns and tables in the database. Maintaining data consistency of derived columns is a popular use of triggers.

ALTER TABLE inventory
ADD COLUMN lost BOOLEAN DEFAULT FALSE AFTER store\_id;

ALTER TABLE inventory
ADD COLUMN in\_stock BOOLEAN DEFAULT TRUE AFTER lost;

To reverse the operations:

ALTER TABLE inventory DROP COLUMN lost;
ALTER TABLE inventory DROP COLUMN in\_stock;

To populate the columns initially:

UPDATE inventory
SET lost = FALSE;

UPDATE inventory
SET in\_stock = inventory\_in\_stock(inventory\_id);

-- checking
select \* from inventory limit 5;

 We now need to define triggers when we insert or update a rental row. We assume that a rental row will never be deleted.

The trigger after inserting a rental row is shown below. There may be an error. Can you spot it?

DELIMITER $$
CREATE TRIGGER insert\_rental AFTER INSERT ON rental FOR EACH ROW
BEGIN
   IF (new.return\_date IS NULL)
   THEN
      UPDATE inventory
      SET in\_stock = FALSE
      WHERE inventory.inventory\_id = rental.inventory\_id;
   END IF;
END$$
DELIMITER ;

Note:

* The trigger name, which can be used later.
* The event that activates the trigger: "AFTER INSERT ON rental".
* The "FOR EACH ROW" clause to indicate the action will be conducted for each new rental row insertion.
* The keyword "new" is used to indicate the row *after* the activating event. The keyword old is used to indicate the row *before* the activating event
* The error in the code above: rental.inventory\_id should be new.inventory\_id

Checking:

--  Checking insert trigger.
SELECT \* FROM rental WHERE inventory\_id = 1;
SELECT \* FROM inventory WHERE inventory\_id = 1;

--  Rent inventory\_id 1 out.
INSERT INTO rental
VALUES (16050,'2011-11-11 22:53:30',1,130,NULL,1,'2011-11-11 23:30:53');

SELECT \* FROM rental WHERE inventory\_id = 1;
SELECT \* FROM inventory WHERE inventory\_id = 1;

For the update trigger:

DELIMITER $$
CREATE TRIGGER update\_rental AFTER UPDATE ON rental FOR EACH ROW
BEGIN
   IF ((old.return\_date IS NULL) and (new.return\_date IS NOT NULL))
   THEN
      UPDATE inventory
      SET in\_stock = TRUE
      WHERE inventory.inventory\_id = new.inventory\_id;
   END IF;
END$$
DELIMITER ;

To check:

--  Checking insert trigger.
SELECT \* FROM rental WHERE inventory\_id = 1;
SELECT \* FROM inventory WHERE inventory\_id = 1;

UPDATE rental
SET return\_date = '2011-11-11 23:30:53'
WHERE rental\_id = 16050;

SELECT \* FROM rental WHERE inventory\_id = 1;
SELECT \* FROM inventory WHERE inventory\_id = 1;

--  House keeping to revert to original state
DELETE FROM rental WHERE rental\_id = 16050;
DROP TRIGGER insert\_rental;
DROP TRIGGER update\_rental;
ALTER TABLE inventory DROP COLUMN lost;
ALTER TABLE inventory DROP COLUMN in\_stock;

* Bad database designs:
	1. Unnecessary redundancy: inefficient storage
	2. Anomaly: conflicts in data, difficulties in maintenance.

We should avoid all kinds of redundancy. No.

Redundancy can be very useful and important: error checking and recovery. Fault tolerance. Performance.

***Example:***

Consider the relation

EMPLOYEE(EMP\_NO, NAME, DEPT\_NO, MANAGER\_NO).

Assumptions made:

1. Every employee works for only one department.
2. Every department has only one manager.
3. Every manager manages only one department.
4. Every employee is represented as a tuple in the EMPLOYEE relation.
5. Every employee has an unique EMP\_NO.

Thus, EMP\_NO is a candidate key.

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** | **MANAGER\_NO** |
| 101 | Lady Gaga | *D123* | *110* |
| 122 | Brad Pitts | *D123* | *110 -> 555?* |
| 140 | Lebron James | *D123* | *110* |
| 155 | Narendra Modi | D225 | 205 |
| 167 | Jennifer Lopez | D225 | 205 |
| 311 | John Smiths | D337 | 333 |

**Problem:**

* Unnecessary redundancy: the fact Manager 110 manages department D123 is stored three times.

**Update Anomaly:**

(a) 415 is the new manager of department D123

* Inefficiency in update.
* Potential inconsistency.

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** | **MANAGER\_NO** |
| 101 | Lady Gaga | D123 | **415** |
| 122 | Brad Pitts | D123 | **415** |
| 140 | Lebron James | D123 | ***110 if not updated*** |
| 155 | Narendra Modi | D225 | 205 |
| 167 | Jennifer Lopez | D225 | 205 |
| 311 | John Smiths | D337 | 333 |

SQL: it works but not efficient.

UPDATE employee
SET MANAGER\_NO = 415
WHERE DEPT\_NO = 'D123';

(b) Jennifer Lopez changes to working for department D337:

* Need to know the manager of D337.
* Potential inconsistency.

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** | **MANAGER\_NO** |
| 101 | Lady Gaga | D123 | 110 |
| 122 | Brad Pitts | D123 | 110 |
| 140 | Lebron James | D123 | 110 |
| 155 | Narendra Modi | D225 | 205 |
| 167 | Jennifer Lopez | **D337** | *205 if not updated* |
| 311 | John Smiths | D337 | 333 |

The command:

UPDATE Employee
SET DEPT\_NO = 'D337'
WHERE NAME = 'Jennifer Lopez';

will produces inconsistent result.

You need to update both DEPT\_NO and MANAGER\_NO. However,

UPDATE Employee
SET DEPT\_NO = 'D337',
     MANAGER\_NO = (SELECT DISTINCT MANAGER\_NO FROM Employee WHERE DEPT\_NO = 'D337')
WHERE NAME = 'Jennifer Lopez';

will not work in MySQL as one cannot include a SELECT clause on the same table in the SET clause.

In MysqL:

SELECT DISTINCT MANAGER\_NO INTO @dept
FROM Employee WHERE DEPT\_NO = 'D337';

SET @dept = 100; -- semantic error

UPDATE Employee
SET DEPT\_NO = 'D337',
     MANAGER\_NO = @dept
WHERE NAME = 'Jennifer Lopez';

**Insertion Anomaly:**

Creating a new department D777, with manager 520, currently with no employee is not possible.

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** | **MANAGER\_NO** |
| 101 | Lady Gaga | D123 | 110 |
| 122 | Brad Pitts | D123 | 110 |
| 140 | Lebron James | D123 | 110 |
| 155 | Narendra Modi | D225 | 205 |
| 167 | Jennifer Lopez | D225 | 205 |
| 311 | John Smiths | D337 | 333 |
| **????** | **????** | **D777** | **520*(cannot be added)*** |

**Deletion Anomaly:**

John Smiths no longer works here. Result: the information that 333 is the manager of department D337 is lost. (Not in 3rd normal form)

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** | **MANAGER\_NO** |
| 101 | Lady Gaga | D123 | 110 |
| 122 | Brad Pitts | D123 | 110 |
| 140 | Lebron James | D123 | 110 |
| 155 | Narendra Modi | D225 | 205 |
| 167 | Jennifer Lopez | D225 | 205 |

A standard way of resolving these issues are by proper **decomposition**: breaking down a relation into two or more relations.

Example: decomposition into two relations:

EMP(EMP\_NO, NAME, DEPT\_NO): in 3NF, BCNF

|  |  |  |
| --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** |
| 101 | Lady Gaga | D123 |
| 122 | Brad Pitts | D123 |
| 140 | Lebron James | D123 |
| 155 | Narendra Modi | D225 |
| 167 | Jennifer Lopez | ~~D225~~ D337 |
| ~~311~~ | ~~John Smiths~~ | ~~D337~~ |

DEPARTMENT(DEPT\_NO, MANAGER\_NO): in 3NF, BCNF

|  |  |
| --- | --- |
| **DEPT\_NO** | **MANAGER\_NO** |
| D123 | ~~110~~ 315 |
| D225 | 205 |
| D337 | 333 |
| D777 | 520 |

To obtain the original relation EMPLOYEE(EMP\_NO, NAME, DEPT\_NO, MANAGER\_NO) from

EMP(EMP\_NO, NAME, DEPT\_NO)
DEPARTMENT(DEPT\_NO, MANAGER\_NO)

Relational algebra:

EMPLOYEE = EMP |x| DEPARTMENT (lossless decomposition)

SQL:

SELECT EMP.\*, DEPARTMENT.MANAGER\_NO INTO EMPLOYEE
FROM EMP, DEPARTMENT
WHERE EMP.DEPT\_NO = DEPARTMENT.DEPT\_NO;

1. There is no loss of information.
2. No previously mentioned redundancy and anomaly.

**Methods for good designs**

* Integrity Rules:  Constraints for avoiding impossible/illegal/incorrect configurations.
* Normal Forms:  A set of rules for designing good relation schemas.

**Integrity Rules:**

* Most of the integrity rules are application-dependent.
* Need to analyze the semantic of the applications to find out the integrity rules.
* These are known as *Database-Specific Integrity Rules*.

***Examples***:

1. Student Id should be a seven digit number
2. Date of Birth should be greater than 1900.
3. The room number of Delta Building should start with a 'D'.
4. A student cannot take more than 24 credits in any semester.
5. A student must show proof of meningitis shot before registration for the first semester.

**General Integrity Rules:**

* Should be satisfied by *every* database, but not necessarily enforced by every DBMS package.
* *Entity Integrity Rule* and *Referential Integrity Rule*: based on the concepts of primary keys, superkeys, and foreign keys

**Entity Integrity Rule**

* Entity Integrity: no*component* of a *candidate key* of a relation can have a null value.
* Meaning: In a relational database, a tuple that cannot be identified will not be stored.

***Example:***

EMPLOYEE(EMP\_NO, NAME, DEPT\_NO, SALARY)

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** | **SALARY** |
| 101 | Lady Gaga | D123 | 550000000 |
| 122 | Brad Pitts | D123 | 101000 |
| 140 | Lebron James | D123 | 50000000 |
| 155 | Narendra Modi | @ | @ |
| *@* | Jennifer Lopez | D225 | 2000000***(cannot be added)*** |
| *@* | John Smiths | D337 | 70000***(cannot be added)*** |

* If EMP\_NO is a candidate key, this EMPLOYEE instance does not satisfy the entity integrity rule.
* If we accept the relation instance above as valid, then EMP\_NO cannot be a candidate key.

**Referential Integrity Rule**

* Referential integrity rule: the database does not contain any *unmatched* *non-null foreign key* values.
* Any non-null value of a foreign key K must appear in the base (host) relation where K is a candidate key.

***Example:***

EMP(EMP\_NO, NAME, DEPT\_NO)

|  |  |  |
| --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** |
| 101 | Lady Gaga | D123 |
| 122 | Brad Pitts | D123 |
| 140 | Lebron James | *@: maybe ok.* |
| 155 | Narendra Modi | D225 |
| 167 | Jennifer Lopez | D225 |
| 311 | John Smiths | D337 |

DEPARTMENT(DEPT\_NO, MANAGER\_NO)

|  |  |
| --- | --- |
| **DEPT\_NO** | **MANAGER\_NO** |
| D123 | 110 |
| D225 | 205 |
| D337 | 333 |

* DEPT\_NO is a foreign key in EMP, referencing DEPT\_NO in DEPARTMENT.
* The referential integrity rule is satisfied.
* Note that DEPT\_NO may be null in EMP.

***Example:***

EMP(EMP\_NO, NAME, DEPT\_NO)

|  |  |  |
| --- | --- | --- |
| **EMP\_NO** | **NAME** | **DEPT\_NO** |
| 101 | Lady Gaga | D123 |
| 122 | Brad Pitts | D123 |
| 140 | Lebron James | D123 |
| 155 | Narendra Modi | D225 |
| 167 | Jennifer Lopez | D225 |
| 311 | John Smiths | D337 |
| **350** | **Bun Yue** | **D119*(may not be added)*** |

DEPARTMENT(DEPT\_NO, MANAGER\_NO)

|  |  |
| --- | --- |
| **DEPT\_NO** | **MANAGER\_NO** |
| D123 | 110 |
| D225 | 205 |
| D337 | 333 |

* The referential integrity rule is not satisfied.

Note:

1. In practical DBMS, pay attention when the referential integrity rule is enforced.
2. For example, in MySQL, only the INNODB data engine may enforce the referential integrity rule.
3. If the DBMS does not enforce the referential integrity rule, it will be the task of the DB developer to do so.

**1. Functional Dependencies**

* Normal forms: a set of rules to avoid redundancy and inconsistency.
* Require the concepts of data dependencies. Examples:
	1. functional dependency (FD, most important: up to BCNF)
	2. multivalued dependency (MVD for 4NF)
	3. join dependency (5NF)
* Common Normal Forms in ascending order: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF, DKNF, 6NF.
* Higher normal forms are more restrictive.
* A relation is in a higher normal form implies that it is in a lower normal form, but not *vice versa*.

***Example:***

If a relation R is in BCNF, then R is also in 3NF, 2NF and 1NF.

If a relation is in 2NF, then

1. It is in 1NF,
2. it may or may not be in 3NF, and
3. it may or may not be in BCNF.

If a relations is not in 3NF, then

1. It is not in BCNF.
2. It may or may not be in 1NF or 2NF.
* In general, the higher the normal forms a relation is in, the better the design of the relation in terms of avoiding redundancy and inconsistency is.
* However, it may be necessary to consider other issues, especially performance.
	+ Higher normal forms may be achieved by decomposition, resulting in more relations. More joins may then be needed to provide the data for a query, decreasing performance.
* 1NF is usually assumed. However, there are relations not in 1NF in both theory and practice.
	+ For an example, a composite data type may be supported by a specific DBMS vendor.
	+ Standard SQL supports many non-1NF features.
* 2NF are more interesting for *historical* reasons.
* 4NF and 5NF involves the concept of *multivalued* and *join* dependencies (MVD and JD). They are hard to understand and even harder to apply in most situations.
* Domain Key Normal Form (DKNF) involves the concept of constraints.
* Based on the concept of *functional dependencies* (FD), the most important normal forms are
	+ 3NF and
	+ BCNF (*Boyce-Codd Normal Form*).