ECE30030/ITP30010 Database Systems

What We Missed: Concepts Recap

Reading: Chapter 7

Charmgil Hong

charmgil@handong.edu

Spring, 2025

Handong Global University

Announcements

- Quiz #2 on week #14
  - Thursday, June 5
  - SQL DML
    - SELECT … FROM … WHERE …
    - JOIN
    - Advanced SQL DML

Term Project

- Team assignments and problems are announced today

Agenda

- Integrity constraints
- Keys
- Views

Integrity Constraints

- Integrity constraints protect the database against accidental damages
  - Work by ensuring that authorized changes to the database do not result in a loss of data consistency; *e.g.*,
    - A checking account must have a balance greater than $10,000.00
    - A salary of a bank employee must be at least $4.00 an hour
    - A customer must have a (non-null) phone number
  - That is, we can put some restrictions like what values are allowed to be inserted, what kind of modification and deletions are allowed in the relation

Integrity Constraints

- Four types of constraints
  - Domain constraints
  - Key constraints
  - Entity integrity constraints
  - Referential integrity constraints

Domain Constraints

- Domain constraints
  - Every domain must contain atomic values
    - Composite and multi-valued attributes are not allowed
  - Datatype check: Assign a data type to a column, and limit the values that the column can contain
  - The **CHECK** constraint is used to limit the value range that can be placed in a column

CHECK Clause

- The **CHECK** (*P*) clause specifies a predicate *P* that must be satisfied by every tuple in a relation
  - *E.g.*, ensure that semester is one of fall, winter, spring or summer

```
CREATE TABLE section (
    course_id VARCHAR(8),
    sec_id VARCHAR(8),
    semester VARCHAR(6),
    year NUMERIC(4,0),
    building VARCHAR(15),
    room_number VARCHAR(7),
    time_slot_id VARCHAR(4),
    PRIMARY KEY (course_id, sec_id, semester, year),
    CHECK (semester IN ('Fall', 'Winter', 'Spring', 'Summer')))
```

- As of MySQL 8.0.16, the CREATE TABLE supported essential features of table and column CHECK constraints for all storage engines
  - Prior to MySQL 8.0.16, the CHECK constraint is just parsed and ignored

Key Constraints

- Key constraints = Uniqueness constraints
  - Ensure that every tuple in the relation should be unique
  - Null values are not allowed in PRIMARY KEYs, hence NOT NULL constraint is also a part of key constraint (entity integrity constraint)

  - Relevant SQL specifiers
    - **PRIMARY KEY**
    - **UNIQUE**
    - **NOT NULL**

UNIQUE Constraints

- **UNIQUE** (A_1, A_2, …, A_m)
  - The UNIQUE specification states that the attributes A_1, A_2, …, A_m form a candidate key
  - Candidate keys are permitted to be null (in contrast to primary keys)

  - *E.g.*, **CREATE TABLE** suppliers (

```
supplier_id INT AUTO_INCREMENT,
name VARCHAR(255),
phone VARCHAR(15) NOT NULL UNIQUE,
address VARCHAR(255),
PRIMARY KEY (supplier_id),
UNIQUE (name, address));
```

NOT NULL Constraints

- **NOT NULL**
  - *E.g.*, Declare *name* and *budget* to be NOT NULL
    - *name* **VARCHAR**(20) **NOT NULL**
      budget **NUMERIC**(12,2) **NOT NULL**

Entity Integrity Constraints

- Entity integrity constraints: Each tuple in a relation must be uniquely identifiable
  - Enforced through the primary key
    - **UNIQUE**: No two rows can have the same primary key value
    - **NOT NULL**: Primary key values cannot be NULL
  - No primary key can take NULL value, since a primary key identifies each tuple uniquely in a relation

Referential Integrity Constraints

- Ensures that a value appearing in one relation for a given set of attributes also appears for a certain set of attributes in another relation
  - *E.g.*, if "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology"
  - Formal definition
    - Let *A* be a set of attributes. Let *R* and *S* be two relations that contain attributes *A* and where *A* is the primary key of *S*
    - *A* is said to be a foreign key of *R* if for any values of *A* appearing in *R* these values also appear in *S*

Referential Integrity Constraints

- Foreign keys can be specified as part of the SQL DDL (CREATE TABLE) statement
  - *E.g.*, **FOREIGN KEY** (*dept_name*) **REFERENCES** *department*

- By default, a foreign key references the primary key attributes of the referenced table
- SQL allows a list of attributes to be referenced specified explicitly
  - *E.g.*, **FOREIGN KEY** (*dept_name*) **REFERENCES** *department* (*dept_name*)

Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation

- An alternative, in case of delete or update is to cascade
  - *E.g.*, **CREATE TABLE** *course* (

```
(…
 dept_name VARCHAR(20),
 FOREIGN KEY (dept_name) REFERENCES department
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  . . .)
```

  - Instead of cascade we can use:
    - **SET NULL**
    - **SET DEFAULT**

Integrity Constraint Violation

- Example:

CREATE TABLE *person* (
ID **CHAR**(10),
name **CHAR**(40),
mother **CHAR**(10),
father **CHAR**(10),
PRIMARY KEY *ID,*
FOREIGN KEY *father* **REFERENCES** *person,*
FOREIGN KEY *mother* **REFERENCES** *person*)

- How to insert a tuple without causing constraint violation?
    - Insert *father* and *mother* of a person before inserting *person*
    - OR, set *father* and *mother* to null initially, update after inserting all *persons* (not possible if *father* and *mother* attributes declared to be **NOT NULL**)
    - OR defer constraint checking

Agenda

- Integrity constraints
- **Keys**
- Views

Keys

- Key: An attribute or a set of attributes, which help(s) uniquely identify a tuple of data in a relation

EmployeeID	Name	Branch	Email
10201	Cooper	DBMI	cooper@institute.edu
10203	Abraham	DBMI	laboriel@institute.edu
10204	Abraham	CS	abe@institute.edu
10207	Elly	EE	elly@institute.edu

- Q: Which of the attributes can be a key?

Keys

- Key: An attribute or a set of attributes, which help(s) uniquely identify a tuple of data in a relation
  - Why we need keys?
    - To force identity of data and
    - To ensure integrity of data is maintained
    - To establish relationship between relations
  - Types of Keys
    - Super key
    - Candidate key
    - Primary key
    - Alternate key
    - Foreign key
    - Composite key
    - Compound key
    - Surrogate key

Super Keys

- Any possible unique identifier
- Any attribute or any set of attributes that can be used to identify tuple of data in a relation; *i.e.*, any of
  - Attributes with unique values or
  - Combinations of the attributes
  - *E.g.*,

EmployeeID	FileCD	Name	Branch	Email
10201	D-201-C	Cooper	DBMI	cooper@institute.edu
10203	D-203-A	Abraham	DBMI	laboriel@institute.edu
10204	C-204-A	Abraham	CS	abe@institute.edu
10207	E-207-E	Elly	EE	elly@institute.edu

Candidate Keys

- Minimal subset of super key
  - If any proper subset of a super key is also a super key, then that (super key) cannot be a candidate key
  - *E.g.,*

EmployeeID	FileCD	Name	Branch	Email
10201	D-201-C	Cooper	DBMI	cooper@institute.edu
10203	D-203-A	Abraham	DBMI	laboriel@institute.edu
10204	C-204-A	Abraham	CS	abe@institute.edu
10207	E-207-E	Elly	EE	elly@institute.edu

➔ *EmployeeID* *EmployeeID + FileCD* *EmployeeID + FileCD + Email*

FileCD *EmployeeID + Email*

Email *FileCD + Email*

Primary Keys (PKs)

- The candidate key chosen to uniquely identify each row of data in a relation
  - No two rows can have the same PK value
  - PK value cannot be NULL (every row must have a primary key value)

EmployeeID	FileCD	Name	Branch	Email
10201	D-201-C	Cooper	DBMI	cooper@institute.edu
10203	D-203-A	Abraham	DBMI	laboriel@institute.edu
10204	C-204-A	Abraham	CS	abe@institute.edu
10207	E-207-E	Elly	EE	elly@institute.edu

➔ *EmployeeID*

FileCD ***Pick any one as PK***

Email

Alternate Keys

- The candidate keys that are NOT chosen as PK in a relation

EmployeeID	FileCD	Name	Branch	Email
10201	D-201-C	Cooper	DBMI	cooper@institute.edu
10203	D-203-A	Abraham	DBMI	laboriel@institute.edu
10204	C-204-A	Abraham	CS	abe@institute.edu
10207	E-207-E	Elly	EE	elly@institute.edu

➔ *EmployeeID*

FileCD

Email

If we choose EmployeeID as PK,
then FileCD and Email become alternate keys

Foreign Keys

- An attribute in a relation that is used to define its relationship with another relation
  - Using foreign key helps in maintaining data integrity for tables in relationship

Employee

EmployeeID	FileCD	Name	Branch	Email
10201	D-201-C	Cooper	DBMI	cooper@institute.edu
10203	D-203-A	Abraham	DBMI	laboriel@institute.edu
10204	C-204-A	Abraham	CS	abe@institute.edu
10207	E-207-E	Elly	EE	elly@institute.edu

Branch

Branch	Address
DBMI	5607 Baum Blvd
CS	260 S Bouquet St
EE	3700 O'Hara St
BIO	4249 Fifth Ave

Composite & Compound Keys

- Composite key: Any key with more than one attribute
  - *E.g.,*

EmployeeID	FileCD	Name	Branch	Email
10201	D-201-C	Cooper	DBMI	cooper@institute.edu
10203	D-203-A	Abraham	DBMI	laboriel@institute.edu
10204	C-204-A	Abraham	CS	abe@institute.edu
10207	E-207-E	Elly	EE	elly@institute.edu

➔ *EmployeeID + FileCD,* *EmployeeID + Email,* *FileCD + Email*

EmployeeID + FileCD + Email

- Compound key: A composite key that has at least one attribute, which is a foreign key
  - *E.g.,* Let us assume that we have defined a composite key (FileCD, Branch), it is also a compound key (considering the Branch table)

Surrogate Keys

- If a relation has no attribute that can be used as a key, then we create an artificial attribute for this purpose
  - It adds no meaning to the data, but serves the sole purpose of identifying tuples uniquely in a table
    - ○○○○_ID with auto increment

Agenda

- Integrity constraints
- Keys
- **Views**

Views

- It is not always desirable for all users to see the entire logical model of data
  - *E.g.*, consider a user who needs to know an instructor name and department, but not the salary

    ➔ This user only needs to see the following relation (in SQL):
      - **SELECT** *ID*, *name*, *dept_name*
        FROM *instructor*

- View: provides a mechanism to hide certain data from the view of certain users
  - A view is a relation defined in terms of stored tables (called *base tables*) and other views
  - Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a view

Views

- Syntax:

  CREATE VIEW *v* **AS** < query expression >

  where <query expression> is any legal SQL expression, and *v* represents the view name

  - Once a view is defined, the view name can be used to refer to the virtual relation that the view generates
  - View definition is not the same as creating a new relation
  - A view definition causes the saving of an expression; the expression is substituted into queries using the view

View Examples

- A view of instructors without their salary:
  - **CREATE VIEW** *faculty* ***AS***
    SELECT *ID*, *name*, *dept_name*
    FROM *instructor*

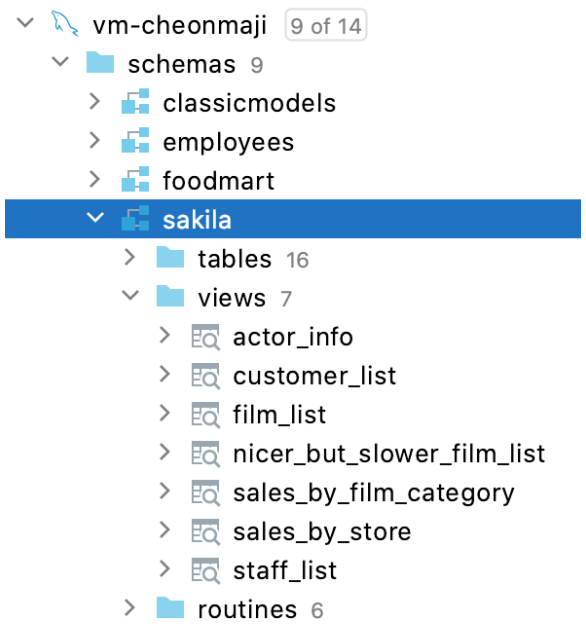
- Querying on a view is also possible:
  - **SELECT** *name*
    FROM *faculty*
    WHERE *dept_name* = 'Biology’

  - *C.f.*, find all instructors in the Biology department:
    - **SELECT** *name*
      FROM *instructor*
      WHERE *dept_name* = 'Biology’

View Examples

- The attribute names of a view can be specified explicitly
  - **CREATE VIEW** *departments_total_salary(dept_name, total_salary)* **AS**
    SELECT *dept_name*, **SUM**(*salary*)
    FROM *instructor*
    GROUP BY *dept_name*;
    - Since the expression **SUM**(*salary*) does not have a name, the attribute name is specified explicitly in the view definition

- The *sakila* database, in the Class VM image, includes 7 sample views



View Expansion

- View expansion: A way to define the meaning of views defined in terms of other views
  - Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations
  - View expansion of an expression repeats the following replacement step:

    repeat

    Find any view relation v_i in e_1

    Replace the view relation v_i by the expression defining v_i

    until no more view relations are present in e_1
  - As long as the view definitions are not recursive, this loop will terminate

Views Defined Using Other Views

- One view may be used in the expression defining another view
  - A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
  - A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
  - A view relation v is said to be *recursive* if it depends on itself

Views Defined Using Other Views

- Examples
  - **CREATE VIEW** *physics_fall_2017* **AS**
    SELECT *course.course_id, sec_id, building, room_number*
    FROM *course, section*
    WHERE *course.course_id* = *section.course_id*
    AND *course.dept_name* = 'Physics'
    AND *section.semester* = 'Fall'
    AND *section.year* = '2017';

  - **CREATE VIEW** *physics_fall_2017_watson* **AS**
    SELECT *course_id, room_number*
    FROM *physics_fall_2017*
    WHERE *building*= 'Watson';

Views Defined Using Other Views

- Both queries are equivalent (view expansion):
  - **CREATE VIEW** *physics_fall_2017_watson* **AS**
    SELECT *course_id, room_number*
    FROM *physics_fall_2017*
    WHERE *building*= 'Watson';

  - **CREATE VIEW** *physics_fall_2017_watson* **AS**
    SELECT *course_id, room_number*
    FROM (**SELECT** *course.course_id, sec_id, building, room_number*
    FROM *course, section*
    WHERE *course.course_id = section.course_id*
    AND *course.dept_name* = 'Physics'
    AND *section.semester* = 'Fall'
    AND *section.year* = '2017')
    WHERE *building*= 'Watson';

Materialized Views

- Two kinds of views
  - Virtual: not stored in the database; just a query for constructing the relation
  - Materialized: physically constructed and stored

- Materialized view: pre-calculated (materialized) result of a query
  - Unlike a simple VIEW the result of a Materialized View is stored somewhere, generally in a table
  - Used when:
    - Immediate response is needed
    - The query where the Materialized View bases on would take to long to produce a result
  - Materialized Views must be refreshed occasionally

- MySQL does NOT support materialized views

Update via a View

- Add a new tuple to *faculty* view which we defined earlier
  INSERT INTO *faculty* **VALUES** ('30765', 'Green', 'Music');
  - This insertion must be represented by the insertion into the *instructor* relation
    - Must have a value for salary

  - Must have a value for salary
    1) Reject the insert, OR
    2) Inset the tuple ('30765', 'Green', 'Music', null) into the *instructor* relation

Update via a View

- Some updates cannot be translated uniquely
  - *E.g.*, **CREATE VIEW** *instructor_info* **AS**
    SELECT *ID*, *name*, *building*
    FROM *instructor*, *department*
    WHERE *instructor*.*dept_name* = *department*.*dept_name*;

    then, **INSERT INTO** *instructor_info*
    VALUES ('69987', 'White', 'Taylor');

  - Issues
    - Which department, if multiple departments are in Taylor?
    - What if no department is in Taylor?

    - On MySQL, an "*SQL error (1394): Can not insert into join view without fields list*" occurs

Update via a View

- Example
  - **CREATE VIEW** *history_instructors* **AS**
    SELECT *
    FROM *instructor*
    WHERE *dept_name*='History';
- What happens if one inserts ('25566', 'Brown', 'Biology', 100000) into *history_instructors*?
  - **INSERT INTO** history_instructors
    VALUES ('25566', 'Brown', 'Biology', 100000)

ID	name	dept_name	salary
32343	El Said	History	60000.00
58583	Califieri	History	62000.00

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
25566	Brown	Biology	100000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Update via a View

- Most SQL implementations allow updates only on simple views
  - The **FROM** clause has only one database relation
  - The **SELECT** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **DISTINCT** specification
  - Any attribute not listed in the **SELECT** clause can be set to null
  - The query does not have a **GROUP BY** or **HAVING** clause