

# Introduction to Data Management

## CSE 344

### Lecture 16: Constraints

# Announcements

- WQ5 due tonight
  - deadline extended only for this WQ
- HW4 due tomorrow
- HW 5 posted, due next Thursday (02/20)
- No class/office hour on Monday, February 17 (President's day)
- Midterm: Wednesday, February 19, in class

# Midterm

- All material up to and including Lecture 15
  - SQL, basic evaluation + indexes, RA, datalog-with-negation, RC, XML/XPath/XQuery, E/R diagram
- Open books, open notes
  - Don't waste paper printing stuff. Normally, you shouldn't need any notes during the exam. My suggestion is to print, say, 5-6 selected slides from the lecture notes that you had trouble with, and to print your own homework, just in case you forget some cool solution you used there.
  - Make sure you understand all the concepts!

# Where We Are?

We are learning about database design

- How to design a database schema?
- Last time: Real world -> E/R Diagrams -> Relations

Next, we will learn more about **good** schemas

- Today: Constraints and data integrity
  - Reading: 7.1, 7.2, 7.4, 7.5
- Next time: Schema normalization, then Views

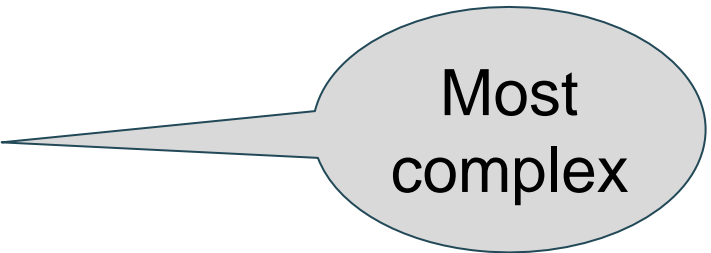
# Constraints in SQL

## Constraints in SQL:

- **Keys, foreign keys**
  - you already know these!
- **Attribute-level** constraints
- **Tuple-level** constraints
- **Global** constraints: assertions



simplest



Most  
complex

- The more complex the constraint, the harder it is to check and to enforce

# Key Constraints

Product(name, category)

```
CREATE TABLE Product (  
    name CHAR(30) PRIMARY KEY,  
    category VARCHAR(20))
```

OR:

```
CREATE TABLE Product (  
    name CHAR(30),  
    category VARCHAR(20)  
PRIMARY KEY (name))
```

How can we specify both name and category as PK?

# Keys with Multiple Attributes

Product(name, category, price)

```
CREATE TABLE Product (  
    name CHAR(30),  
    category VARCHAR(20),  
    price INT,  
    PRIMARY KEY (name, category))
```

Name	Category	Price
Gizmo	Gadget	10
Camera	Photo	20
Gizmo	Photo	30
<del>Gizmo</del>	<del>Gadget</del>	<del>40</del>

How can we specify more than one keys?

# Other Keys

```
CREATE TABLE Product (  
    productID CHAR(10),  
    name CHAR(30),  
    category VARCHAR(20),  
    price INT,  
    PRIMARY KEY (productID),  
    UNIQUE (name, category))
```

There is at most one **PRIMARY KEY**;  
there can be many **UNIQUE**

How can we specify foreign keys?



# Foreign Key Constraints

```
CREATE TABLE Purchase (  
  prodName CHAR(30)  
  REFERENCES Product(name),  
  date DATETIME)
```

Referential  
integrity  
constraints

prodName is a **foreign key** to Product(name)  
name must be a **key** in Product

May write  
just  
Product  
if name is  
PK

# Foreign Key Constraints

Product

<u>Name</u>	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
CREATE TABLE Purchase (  
    prodName CHAR(30)  
    REFERENCES Product(name),  
    date DATETIME)
```

# Foreign Key Constraints

- Example with multi-attribute primary key

```
CREATE TABLE Purchase (  
    prodName CHAR(30),  
    category VARCHAR(20),  
    date DATETIME,  
    FOREIGN KEY (prodName, category)  
    REFERENCES Product(name, category)
```

- (name, category) must be a KEY in Product

# What happens during updates ?

Types of updates:

- In Purchase: insert/update
- In Product: delete/update

Can you think of some options?

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

# What happens during updates ?

- SQL has three policies for maintaining referential integrity:
- Reject violating modifications (default)
- Cascade: after delete/update do delete/update
- Set-null set foreign-key field to NULL
  - results in dangling tuples, that won't participate in joins
- Which policy to use? Depends on the application

# Maintaining Referential Integrity

```
CREATE TABLE Purchase (  
    prodName CHAR(30),  
    category VARCHAR(20),  
    date DATETIME,  
    FOREIGN KEY (prodName, category)  
        REFERENCES Product(name, category)  
        ON UPDATE CASCADE  
        ON DELETE SET NULL )
```

# Constraints on Attributes and Tuples

- Constraints on attributes:
  - NOT NULL** -- obvious meaning...
  - CHECK** condition -- any condition !
- Constraints on tuples
  - CHECK** condition
- NOT NULL:
  - (i) we cannot insert a tuple with null attribute value, and
  - (ii) cannot use SET NULL on update

# Constraints on Attributes and Tuples

Attribute-based

```
CREATE TABLE R (  
  A int NOT NULL,  
  B int CHECK (B > 50 and B < 100),  
  C varchar(20),  
  D int,  
  CHECK (C >= 'd' or D > 0))
```

Tuple-based



# CHECK condition

- Attribute-based
  - is checked whenever (and only when) any tuple gets a new value of that attribute by UPDATE or INSERT
  - can involve only one attribute
- Tuple-based
  - is checked every time a tuple is inserted/updated in that relation
  - can involve one or more attributes
- For both,
  - modification is rejected if condition is not satisfied
  - Can also use other relations

(e.g. attr-based) **B int CHECK B IN (SELECT C FROM R)**

# Constraints on Attributes and Tuples

```
CREATE TABLE Product (  
    productID CHAR(10),  
    name CHAR(30),  
    category VARCHAR(20),  
    price INT CHECK (price > 0),  
    PRIMARY KEY (productID),  
    UNIQUE (name, category))
```

# Constraints on Attributes and Tuples

What does this constraint do?

```
CREATE TABLE Purchase (  
    prodName CHAR(30)  
    CHECK (prodName IN  
        (SELECT Product.name  
         FROM Product),  
    date DATETIME NOT NULL)
```

What  
is the difference from  
Foreign-Key ?

# Constraints on Attributes and Tuples

What does this constraint do?

```
CREATE TABLE Purchase (  
    prodName CHAR(30)  
    CHECK (prodName IN  
        (SELECT Product.name  
         FROM Product),  
    date DATETIME NOT NULL)
```

What  
is the difference from  
Foreign-Key ?

Constraints on attributes are only checked when the value of the attribute changes (so they could potentially be violated by other changes).  
So, unlike a FK, if Product changes, this check won't catch the problem

# General Assertions

```
CREATE ASSERTION myAssert CHECK  
(NOT EXISTS(  
    SELECT Product.name  
    FROM Product, Purchase  
    WHERE Product.name = Purchase.prodName  
    GROUP BY Product.name  
    HAVING count(*) > 200)  
)
```

But most DBMSs do not implement assertions  
Because it is hard to support them efficiently  
Instead, they provide triggers

# Assertion and Trigger

- An assertion is a Boolean-valued expression that must be true all the time
  - Easy for programmers
  - Hard for DBMS to implement
  - DBMS has to deduce whether a change can affect the truthfulness of an assertion
- A trigger is a series of actions associated with certain events (like insert, update, delete)
  - More frequently used

# Database Triggers

- Event-Condition-Action rules
- Event
  - Can be insertion, update, or deletion to a relation
- Condition
  - Can be expressed on DB state before or after event
- Action
  - Perform additional DB modifications
- Trigger is awakened when Event occurs, then it tests the Condition, if satisfied, Action is performed, otherwise nothing happens

# More About Triggers

- Row-level trigger
  - Executes once for each modified tuple
- Statement-level trigger
  - Executes once for all tuples that are modified in a SQL statement



# Database Triggers Example

When Product.price is updated, if it is decreased then  
set Product.category = 'On sale'

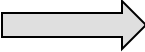
# Database Triggers Example

When Product.price is updated, if it is decreased then set Product.category = 'On sale'

```
CREATE TRIGGER ProductCategories
AFTER UPDATE OF price ON Product
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.price > NewTuple.price)
    UPDATE Product
    SET category = 'On sale'
    WHERE productID = OldTuple.productID
```

# Database Triggers Example

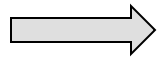
CREATE TRIGGER statement



```
CREATE TRIGGER ProductCategories
AFTER UPDATE OF price ON Product
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.price > NewTuple.price)
    UPDATE Product
    SET category = 'On sale'
    WHERE productID = OldTuple.productID
```

# Database Triggers Example

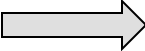
- Whether the trigger uses the db state BEFORE or AFTER the triggering event
- For BEFORE, if condition is true, the triggering event is executed irrespective of whether the condition still holds



```
CREATE TRIGGER ProductCategories
AFTER UPDATE OF price ON Product
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.price > NewTuple.price)
    UPDATE Product
    SET category = 'On sale'
    WHERE productID = OldTuple.productID
```

# Database Triggers Example

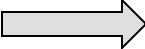
- To refer to the tuple being modified
- Can give name to old or new ROW (tuple) or TABLE
- Options: INSERT, DELETE, UPDATE (OF)
- OLD (resp. NEW) ROW is disallowed for INSERT (resp. DELETE)



```
CREATE TRIGGER ProductCategories
AFTER UPDATE OF price ON Product
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.price > NewTuple.price)
    UPDATE Product
    SET category = 'On sale'
    WHERE productID = OldTuple.productID
```

# Database Triggers Example

- Whether the trigger executes once for each modified row or once for all the modifications made by the SQL statement
- FOR EACH ROW vs. STATEMENT



```
CREATE TRIGGER ProductCategories
AFTER UPDATE OF price ON Product
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.price > NewTuple.price)
    UPDATE Product
    SET category = 'On sale'
WHERE productID = OldTuple.productID
```

# Database Triggers Example

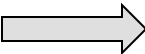
“Condition” to be tested

```
CREATE TRIGGER ProductCategories
AFTER UPDATE OF price ON Product
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple
FOR EACH ROW
→ WHEN (OldTuple.price > NewTuple.price)
    UPDATE Product
    SET category = 'On sale'
    WHERE productID = OldTuple.productID
```

# Database Triggers Example

- “Action” with one or more SQL statements
- Multiple SQL statements are separated by ; and all within BEGIN.. END

```
CREATE TRIGGER ProductCategories
AFTER UPDATE OF price ON Product
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.price > NewTuple.price)
    UPDATE Product
    SET category = 'On sale'
    WHERE productID = OldTuple.productID
```





# SQL Server Example

```
CREATE TRIGGER ProductCategory
ON Product
AFTER UPDATE
AS
BEGIN
    UPDATE Product
    SET category='sale' WHERE productID IN
    (SELECT i.productID from inserted i, deleted d
    WHERE i.productID = d.productID
    AND i.price < d.price)
END
```

# Summary

- Both constraints and triggers are tools that help us keep the database consistent, have their pros and cons
- We learnt
  - Key/Referential Integrity constraints
  - Attribute-based CHECK constraints
  - Tuple-based CHECK constraints
  - Assertions
  - Triggers