Chapter 7

Chapter 7: Constraints

John Connor

March 11, 2019

Not-Null Constraint

Perhaps the simplest constraint is the not-null constraint.

Not-Null Constraint

Perhaps the simplest constraint is the not-null constraint.

If an attribute is constrained to be not-null, then each tuple in the relation must have a value for the attribute.

Not-Null Constraint

Perhaps the simplest constraint is the not-null constraint.
If an attribute is constrained to be not-null, then each tuple in the relation must have a value for the attribute.
For example:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

Keys and Foreign Keys

A key for a relation is a set of attributes such that no two distinct tuples in the relation have the same values for all of attributes in the key.

Keys and Foreign Keys

A key for a relation is a set of attributes such that no two distinct tuples in the relation have the same values for all of attributes in the key.
A foreign-key constraint asserts that a value appearing in one relation must also appear in the primary-key of another relation.

Primary Key Details

In theory, a relation can have many keys. In SQL, a key is selected to be the *primary key* of the relation.

Primary Key Details

In theory, a relation can have many keys. In SQL, a key is selected to be the *primary key* of the relation.

For example:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

where `PK_Person` is just an arbitrary name for the constraint.

Primary Key Details

A nicer looking but potentially less portable statement:

Primary Key Details

A nicer looking but potentially less portable statement: For example:

```
CREATE TABLE Persons (
    ID int PRIMARY KEY,
    LastName varchar(255) PRIMARY KEY,
    FirstName varchar(255),
    Age int
);
```

Unique Columns

You can still have other sets of attributes be unique by using the `UNIQUE` constraint.

Unique Columns

You can still have other sets of attributes be unique by using the UNIQUE constraint.

For example:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (Age,FirstName),
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

(Although this example doesn't really make much sense.)

Foreign Key Details

The attributes referenced by the foreign key constraint must be declared `UNIQUE` or be part of the `PRIMARY KEY` for the relation.

Foreign Key Details

The attributes referenced by the foreign key constraint must be declared `UNIQUE` or be part of the `PRIMARY KEY` for the relation. For example, consider the relations

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

where `presC#` references `cert#`.

Foreign Key Details

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT,
    FOREIGN KEY (pres#) REFERENCES MovieExec(cert#)
);
```

Foreign Key Details

```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presC# INT,
        FOREIGN KEY (pres#) REFERENCES MovieExec(cert#)
 );
```

The slightly nicer but potentially less portable syntax is

```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presC# INT REFERENCES MovieExec(cert#)
 );
```

Enforcing Foreign Keys Constraints

The DBMS will not allow any of the following action

1. Inserting a new Studio tuple with presC# component not NULL and not the cert# component of any MovieExec tuple.
2. Updating a Studio tuple to change the presC# value to a non NULL which is not the cert# component of any MovieExec tuple.
3. Deleting a MovieExec tuple where the non NULL cert# value appears in the pres# component of a tuple in the Studio relation.
4. Updating a MovieExec tuple in a way that changes the cert# value of a tuple that appears in the pres# component of a tuple in the Studio relation.

Enforcing Foreign Keys Constraints: Policies

When the modification is to the relation where the foreign-key constraint is declared, the system must reject the modification.

Enforcing Foreign Keys Constraints: Policies

When the modification is to the relation where the foreign-key constraint is declared, the system must reject the modification. If the modification is to the referenced relation, then there are three options in how to handle it:

Enforcing Foreign Keys Constraints: Policies

When the modification is to the relation where the foreign-key constraint is declared, the system must reject the modification. If the modification is to the referenced relation, then there are three options in how to handle it:

1. *The Default Policy.* Reject the change.

Enforcing Foreign Keys Constraints: Policies

When the modification is to the relation where the foreign-key constraint is declared, the system must reject the modification. If the modification is to the referenced relation, then there are three options in how to handle it:

1. *The Default Policy.* Reject the change.
2. *The Cascade Policy.* Mimic the change. That is delete the tuple if the referenced tuple is deleted, update the component to the new value if the referenced component is updated.

Enforcing Foreign Keys Constraints: Policies

When the modification is to the relation where the foreign-key constraint is declared, the system must reject the modification. If the modification is to the referenced relation, then there are three options in how to handle it:

1. *The Default Policy.* Reject the change.
2. *The Cascade Policy.* Mimic the change. That is delete the tuple if the referenced tuple is deleted, update the component to the new value if the referenced component is updated.
3. *The Set-Null Policy.* Set the value of the foreign key components to `NULL` (or reject if there is a not-null constraint).

Enforcing Foreign Keys Constraints: Example

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

Deferred Checking

Notice that if we create a new studio, we must insert its `MovieExec` tuple before its `Studio` tuple.

Deferred Checking

Notice that if we create a new studio, we must insert its `MovieExec` tuple before its `Studio` tuple.

Alternatively, we can insert a `Studio` tuple with a `NULL` certificate component, insert a studio `MovieExec` tuple, and then update the `Studio` tuple.

Deferred Checking

Notice that if we create a new studio, we must insert its MovieExec tuple before its Studio tuple.

Alternatively, we can insert a Studio tuple with a NULL certificate component, insert a studio MovieExec tuple, and then update the Studio tuple.

Another solution is to defer the checking of the constraint until the end of the transaction.

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT
        REFERENCES MovieExec(cert#)
        DEFERRABLE INITIALLY DEFERRED
);
```

Deferred Checking

More generally if `Foo` is the name of a constraint, then we can make it deferrable with the SQL

```
SET CONSTRAINT FOO DEFERRED;
```

Deferred Checking

More generally if Foo is the name of a constraint, then we can make it deferrable with the SQL

```
SET CONSTRAINT FOO DEFERRED;
```

and we can make it not deferrable with

```
SET CONSTRAINT FOO IMMEDIATE;
```

Check Constraints

One of the more versatile constraints is the `CHECK` constraint.

Check Constraints

One of the more versatile constraints is the CHECK constraint.

Example:

```
presC# INT REFERENCES MovieExec(cert#)
           CHECK (presC# >= 100000)
```

Check Constraints

One of the more versatile constraints is the CHECK constraint.

Example:

```
presC# INT REFERENCES MovieExec(cert#)
           CHECK (presC# >= 100000)
```

This will run each time a tuple is in the relation is inserted or modified.

Check Constraints

One of the more versatile constraints is the `CHECK` constraint.

Check Constraints

One of the more versatile constraints is the CHECK constraint.

Example:

```
presC# INT REFERENCES MovieExec(cert#)
           CHECK (presC# >= 100000)
```

Check Constraints

One of the more versatile constraints is the CHECK constraint.

Example:

```
presC# INT REFERENCES MovieExec(cert#)
           CHECK (presC# >= 100000)
```

The expression that comes after the CHECK is anything that would be valid in a WHERE clause.

Check Constraints

One of the more versatile constraints is the CHECK constraint.

Example:

```
presC# INT REFERENCES MovieExec(cert#)
           CHECK (presC# >= 100000)
```

The expression that comes after the CHECK is anything that would be valid in a WHERE clause.

The expression will be evaluated each time a tuple is inserted or updated. If the expression is false, the insert or update will be rejected.

Check Constraints

One of the more versatile constraints is the CHECK constraint.

Example:

```
presC# INT REFERENCES MovieExec(cert#)
           CHECK (presC# >= 100000)
```

The expression that comes after the CHECK is anything that would be valid in a WHERE clause.

The expression will be evaluated each time a tuple is inserted or updated. If the expression is false, the insert or update will be rejected.

The expression will **not** be modified if a relation referenced by the expression is modified!

Check Constraints: Example (pg. 321)

(Note that on this point the book does not paint a very accurate picture of contemporary RDBMs.)

Check Constraints: Example (pg. 321)

(Note that on this point the book does not paint a very accurate picture of contemporary RDBMs.)

Assume that there are no longer any foreign key constraints.

```
presC# INT REFERENCES MovieExec(cert#)
           CHECK (presC# IN (SELECT cert#
                             FROM MovieExec))
```

What happens when we insert a tuple into Studio with a presC# value not in MovieExec.cert#?

Check Constraints: Example (pg. 321)

(Note that on this point the book does not paint a very accurate picture of contemporary RDBMs.)

Assume that there are no longer any foreign key constraints.

```
presC# INT REFERENCES MovieExec(cert#)
           CHECK (presC# IN (SELECT cert#
                             FROM MovieExec))
```

What happens when we insert a tuple into `Studio` with a `presC#` value not in `MovieExec.cert#`?

What happens when we update a tuple in `Studio` with a `presC#` value not in `MovieExec.cert#`?

Check Constraints: Example (pg. 321)

(Note that on this point the book does not paint a very accurate picture of contemporary RDBMs.)

Assume that there are no longer any foreign key constraints.

```
presC# INT REFERENCES MovieExec(cert#)
           CHECK (presC# IN (SELECT cert#
                             FROM MovieExec))
```

What happens when we insert a tuple into Studio with a presC# value not in MovieExec.cert#?

What happens when we update a tuple in Studio with a presC# value not in MovieExec.cert#?

What happens when we delete a tuple from MovieExec that has the same cert# as the presC# component of a tuple in Studio?

Check Constraints: Explanation

Why would `CHECK` constraints be implemented in this way?

Quiz Preparation:

Exercises for Sections 7.1, 7.2