

From schema to physical
tables

Clarification: SQL subsets

- DDL – data definition language
- DML – data manipulation language:
 - INSERT
 - UPDATE
 - DELETE
- Data retrieval language: Structured Query Language
- We just defining tables and filling data, not using query language yet

Creating schema in Postgre

```
DROP SCHEMA IF EXISTS movies_db CASCADE;  
CREATE SCHEMA movies_db;  
SET SEARCH_PATH TO movies_db;
```

- Now you can use regular syntax without prefixing each object by movies_db

Need to define proper data types

- NUMERIC
- CHAR(n)
- VARCHAR (n)
- DATE
- BOOLEAN
- ENUM

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');  
CREATE TABLE person (  
    name text,  
    current_mood mood  
);
```

By Marina Barsky

Data integrity: basic constraints

Lecture 4

Constraints

- Data types are a way to limit the kind of data that can be stored in a table.
- For many applications, however, the constraint they provide is too coarse.

For example, a column containing a **product price** should probably only accept **positive values**. But there is no standard data type that accepts only positive numbers.

Two types of constraints

- **Referential constraints:**
 - PRIMARY KEY,
 - FOREIGN KEY
- **Value constraints:**
 - NULL,
 - UNIQUE,
 - CHECK - certain set of values

Referential integrity
constraints

Keys in SQL

- Keys play an important role in SQL, because specifying the values of key attributes is a way of referring to a unique tuple in a relation.
- Since updates (entered by users of the database) could potentially violate the uniqueness of a key, DBMSs offer to check this.

Primary key

```
CREATE TABLE Movies (  
    title CHAR(40) PRIMARY KEY,  
    year INT,  
    length INT,  
    type CHAR(2)  
);
```

Defined at column level

```
CREATE TABLE Movies (  
    title CHAR(40),  
    year INT,  
    length INT,  
    type CHAR(2),  
    PRIMARY KEY (title, year)  
);
```

Defined at table level

NULL

- *Null* is a special marker to indicate that a data value represents "missing or inapplicable information"
- This should not be confused with a value of 0. A null value indicates a **lack of a value** - a lack of a value is not the same as a value of zero in the same way that a lack of an answer is not the same as an answer of "no".

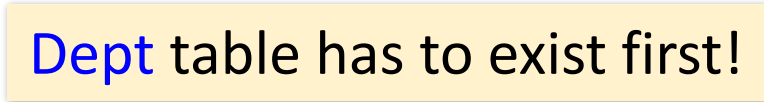
Foreign keys

- In relational model tables are related to each other through common column
- A column (or a set of columns) in one table is a *primary key* of this table, if its value uniquely identifies each tuple (row). Such table is called a **parent table**
- A column in another table that references a primary key column in the parent table is known as a *foreign key*. This table is called a **child table**

Foreign key constraints

Example: Each employee in table Emp must work in a department that is contained in table Dept.

```
CREATE TABLE Emp (  
    empno INT PRIMARY KEY,  
    ... ,  
    deptno INT REFERENCES Dept (deptno)  
);
```



Dept table has to exist first!

Foreign keys: movies

Remark. If you don't specify primary keys or unique constraints in the parent tables, you can't specify foreign keys in the child tables.

```
CREATE TABLE MovieStars (  
    name VARCHAR2(20) PRIMARY KEY,  
    address VARCHAR2(30),  
    gender VARCHAR2(1),  
    birthdate VARCHAR2(20)  
);
```

```
CREATE TABLE Movies (  
    title VARCHAR2(40),  
    year INT,  
    length INT,  
    type VARCHAR2(2),  
    PRIMARY KEY (title, year)  
);
```

```
CREATE TABLE StarsIn (  
    title VARCHAR2(40),  
    year INT,  
    starName VARCHAR2(20),  
    FOREIGN KEY (title, year) REFERENCES Movies (title, year),  
    FOREIGN KEY (starName) REFERENCES MovieStars (name)  
);
```

Self-relationships

- A foreign key constraint may also refer to the same table, i.e., **parent table and child table are identical**.

Example: Every employee must have a manager who must be an employee.

```
CREATE TABLE Emp (  
    empno INT PRIMARY KEY,  
    . . .  
    mgrno INT NOT NULL REFERENCES Emp,  
    . . .  
);
```

Foreign key: may be NULL

Each row in the child table has to satisfy one of the following two conditions:

- Foreign key column value must either
 - appear as a primary key value in the parent table, or
 - be **NULL**

```
CREATE TABLE Emp (  
    empno INT PRIMARY KEY,  
    ... ,  
    deptno INT REFERENCES Dept(deptno)  
);
```

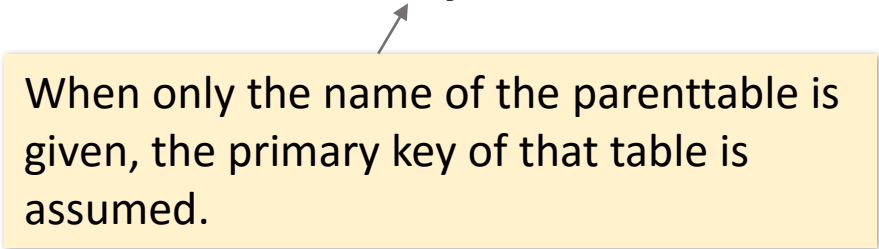
So, for table **Emp**, an employee must not necessarily work in a department, i.e., for the attribute **deptno**, NULL is admissible.

Foreign key: forbidding NULLs

- If we don't want NULL's in a foreign key **we must say so.**

Example: There should always be a project manager, who must be an employee.

```
CREATE TABLE Project (  
    pno INT PRIMARY KEY,  
    pmno INT NOT NULL REFERENCES Emp,  
    ...  
);
```



When only the name of the parenttable is given, the primary key of that table is assumed.

Constraints on deletion (update)

- If there is a foreign-key constraint from table R to S , it is possible that a deletion or update to S causes some tuples of R to “dangle.”

```
CREATE TABLE Beers (  
  name CHAR(20) PRIMARY KEY,  
  manf CHAR(20)  
);
```

```
CREATE TABLE Sells (  
  bar CHAR(20),  
  beer CHAR(20),  
  price REAL,  
  FOREIGN KEY(beer) REFERENCES Beers(name)  
);
```

Enforcing FK constraints

- An insert or update to **Sells** that introduces **a nonexistent beer must be rejected**.
- A deletion or update to **Beers** that removes a beer value found in some tuples of **Sells** can be handled in three ways.
 1. *Default* : Reject the modification (RESTRICT).
 2. *Cascade* : Make the same changes in Sells.
 - Deleted beer: delete Sells tuples referencing that beer.
 - Updated beer: change values in Sells.
 3. *Set NULL* : Change the beer in child table to NULL.

Example

Cascade

- Delete **Bud** from **Beers**:
 - Then delete all tuples from **Sells** that have **beer = 'Bud'**.
- Update the **Bud** tuple by changing 'Bud' to 'Budweiser':
 - Then change all **Sells** tuples with **beer = 'Bud'** so that **beer = 'Budweiser'**.

Set NULL

- Delete the **Bud** tuple from **Beers**:
 - Change all tuples of **Sells** that have **beer = 'Bud'** to have **beer = NULL**.
- Update the **Bud** tuple by changing 'Bud' to 'Budweiser':
 - Same change.

Choosing a Policy

Follow the foreign-key declaration by:

ON [UPDATE, DELETE] [SET NULL, CASCADE]

```
CREATE TABLE Sells (  
    bar          CHAR(20) ,  
    beer         CHAR(20) ,  
    price        REAL ,  
    FOREIGN KEY (beer) REFERENCES Beers (name)  
        ON DELETE SET NULL  
        ON UPDATE CASCADE  
);
```

Constraint names: default

```
CREATE TABLE AB (  
    A PRIMARY KEY NUMBER,  
    B NUMBER  
);
```

- Every constraint, has a name. **Default** (for Postgre): **ab_a_key**.

- To find the constraint names in a table: `\d ab`

...

```
"ab_a_key" PRIMARY KEY, btree (a)
```

Constraint names: explicit

- We can explicitly name the constraint for easier handling.

```
CREATE TABLE AB (  
    A numeric CONSTRAINT my_unique_constraint  
                           PRIMARY KEY,  
    B numeric  
);
```

It can't have the same name as another constraint even if it is in another table.

Dropping/Adding

```
CREATE TABLE AB (  
    A NUMBER,  
    B NUMBER  
);
```

```
ALTER TABLE AB ADD CONSTRAINT my_unique_constraint  
PRIMARY KEY (B);
```

```
ALTER TABLE AB DROP CONSTRAINT  
my_unique_constraint;
```


Chicken and egg problem

```
CREATE TABLE chicken (  
  cID INT PRIMARY KEY,  
  eID INT REFERENCES egg(eID)  
);
```

```
CREATE TABLE egg(  
  eID INT PRIMARY KEY,  
  cID INT REFERENCES chicken(cID)  
);
```

But, if we simply type the above statements, we'll get an error.

- The reason is that the **CREATE TABLE** statement for chicken refers to table egg, which hasn't been created yet!
- Creating egg won't help either, because egg refers to chicken.

Solving chicken/egg: first attempt

- First, create chicken and egg without foreign key declarations.

```
CREATE TABLE chicken(  
  cID INT PRIMARY KEY,  
  eID INT  
);
```

```
CREATE TABLE egg(  
  eID INT PRIMARY KEY,  
  cID INT  
);
```

- Then, add foreign key constraints.

```
ALTER TABLE chicken ADD CONSTRAINT chickenREFegg  
FOREIGN KEY (eID) REFERENCES egg(eID);
```

```
ALTER TABLE egg ADD CONSTRAINT eggREFchicken  
FOREIGN KEY (cID) REFERENCES chicken(cID);
```

They go through...

However, inserting fails...

```
INSERT INTO chicken VALUES (1, 2);
```

*

ERROR: insert or update on table "chicken" violates foreign key constraint "chickenrefegg"

DETAIL: Key (eid)=(2) is not present in table "egg".

```
INSERT INTO egg VALUES (2, 1);
```

*

ERROR: insert or update on table "egg" violates foreign key constraint "eggrefchicken"

DETAIL: Key (cid)=(1) is not present in table "chicken".

Deferrable Constraints

1. We need the ability to group several SQL statements into one atomic unit called *transaction*.
 2. Then, we need a way to tell the SQL system not to check the constraints until the transaction is committed.
- UNIQUE, PRIMARY KEY, and REFERENCES (FK) may be declared “DEFERRABLE” or “NOT DEFERRABLE.”
 - NOT DEFERRABLE is default, means that with every modification the constraint is immediately checked.
 - DEFERRABLE means that we have the **option** of telling the system to wait until a transaction is complete before checking the constraint.

Initially Deferred / Initially Immediate

- If a constraint is deferrable, then we may also declare it as
 - **INITIALLY DEFERRED**, and the check will be deferred to the end of the current transaction.
 - **INITIALLY IMMEDIATE**, (default) and the check will be made before any modification.
- But, because the constraint is *deferrable*, we have the option of later deciding to defer checking:

```
SET CONSTRAINT MyConstraint DEFERRED
```

Setting up solution to chicken/egg

- Here we declare the constraints DEFERRABLE and INITIALLY DEFERRED.

```
ALTER TABLE chicken ADD CONSTRAINT  
  chickenREFegg  
FOREIGN KEY (eID) REFERENCES egg(eID)  
DEFERRABLE INITIALLY DEFERRED;
```

```
ALTER TABLE egg ADD CONSTRAINT eggREFchicken  
FOREIGN KEY (cID) REFERENCES chicken(cID)  
DEFERRABLE INITIALLY DEFERRED;
```

Successful insertions as a single transaction

BEGIN;

INSERT INTO chicken VALUES(1, 2);

INSERT INTO egg VALUES(2, 1);

COMMIT;

Dropping

Finally, to get rid of the tables, we have to drop the constraints first, because we can't drop a table that's referenced by another table.

```
ALTER TABLE egg DROP CONSTRAINT eggREFchicken;  
ALTER TABLE chicken DROP CONSTRAINT  
    chickenREFegg;
```

```
DROP TABLE egg;  
DROP TABLE chicken;
```


Value constraints

Value constraints

- Define if NULL is disallowed
- Define if UNIQUE values are required
- Define CHECK for a set of values

Not Null constraint

```
CREATE TABLE ABC (  
  A numeric NOT NULL,  
  B numeric,  
  C numeric  
);
```

```
insert into ABC values ( 1, null, null);  
insert into ABC values ( 2, 3, 4);  
insert into ABC values (null, 5, 6);
```

The first two records can be inserted, the **third cannot**,
throwing

ERROR: null value in column "a" violates not-null constraint

The **not null** constraint can be altered with:

```
ALTER TABLE ABC ALTER COLUMN A DROP NOT NULL;
```

After this modification, the column A can contain null values.

UNIQUE constraint

```
CREATE TABLE AB (  
  A NUMERIC UNIQUE,  
  B NUMERIC  
);
```

```
insert into AB values (4,5);  
insert into AB values (2,1);  
insert into AB values (6,1);  
insert into AB values (null,9);  
insert into AB values (null,9);  
insert into AB values (2, 7);
```

- The last statement issues

```
ERROR: duplicate key value violates unique  
constraint "ab_a_key"
```

```
DETAIL: Key (a)=(2) already exists.
```

UNIQUE doesn't allow duplicate values in a column. If it encompasses more columns, no two equal combinations are allowed.

However, **nulls can be inserted multiple times.**

UNIQUE on multiple columns

```
CREATE TABLE AB (  
  A NUMBER,  
  B NUMBER,  
  CONSTRAINT UNIQUE (A, B)  
);  
  
insert into AB values (4, 5);  
insert into AB values (4, 1);  
insert into AB values (9, 1);  
insert into AB values (null, null);  
insert into AB values (null, null);  
insert into AB values (null, 9);  
insert into AB values (5, null);  
insert into AB values (5, null);
```

The last statement issues an ERROR

UNIQUE doesn't allow duplicate values in a column. If it encompasses more columns, no two equal combinations are allowed.

However, **nulls can be inserted multiple times.**

So why two combinations (5, null) and (5, null) are not allowed?

Composite **UNIQUE** constraints and **NULL**

- UNIQUE constraint allows more than one NULL values to be inserted: DBMS considers one NULL value is not equal to another NULL value.
- We can insert null values to both columns multiple times. This is because DBMS creates an index for each unique combination, and when all columns are NULL this combination is not included into the index
- But the result changes when we have only one NULL value for the **composite UNIQUE** constraint. In this case the not NULL value is included into the index, and no another tuple with the same combination is allowed

CHECK constraints

allow users to restrict the domain of possible attribute values for columns to admissible ones

```
[CONSTRAINT <name>] CHECK (<condition>)
```

Column-level CHECK constraints: examples

- The name of an employee must consist of upper case letters only;
- The minimum salary of an employee is 500;
- Department numbers must range between 10 and 100:

```
CREATE TABLE Emp (  
    empno NUMBER,  
    ename VARCHAR2(30) CONSTRAINT check_name  
        CHECK( ename = UPPER(ename) ),  
    sal NUMBER CONSTRAINT check_sal  
        CHECK( sal >= 500 ),  
    deptno NUMBER CONSTRAINT check_deptno  
        CHECK( deptno BETWEEN 10 AND 100 )  
);
```


Enforcing CHECK constraints

- DBMS automatically checks the specified conditions each time a database modification is performed on this relation. E.g., the insertion

```
INSERT INTO emp  
VALUES (7999, 'SCOTT', 450, 10) ;
```

causes a constraint violation and it is rejected.

Tuple-level CHECK constraints

- A check constraint can also be a **tuple** constraint, and the <condition> can refer to several columns of the same tuple.
- **Example:**
 - At least two persons must participate in a project, and
 - project's start date must be before project's end date

```
CREATE TABLE Project (  
    ... ,  
    pstart DATE,  
    pend DATE,  
    persons NUMBER CONSTRAINT check_pers CHECK  
    (persons >= 2) ,  
    ... ,  
    CONSTRAINT dates_ok CHECK (pend > pstart)  
);
```

Column constraint

Tuple constraint

Constraint logic

- Create table MovieStar. If the star gender is 'M', then his name must not begin with 'Ms.'.

```
CREATE TABLE MovieStar (  
    name CHAR(20) PRIMARY KEY,  
    address VARCHAR(255),  
    gender CHAR(1),  
    CHECK (gender <> 'M' OR name NOT LIKE 'Ms. %')  
);
```

We can't use an "implication (if)"
We should formulate it in terms of OR.

Checks with sub-queries (theoretically)

Example

```
CREATE TABLE Sells (  
  bar CHAR(20),  
  beer CHAR(20) CONSTRAINT beer_check  
    CHECK ( beer IN (SELECT name FROM Beers) ),  
  price REAL CHECK ( price <= 5.00 )  
);
```

Checks with sub-queries (theoretically)

Such checks are called **assertions**

They state what must be true all the time

DBMS's do not generally support assertions since it is very hard to implement them efficiently

```
CREATE TABLE Sells (  
  bar CHAR(20),  
  price REAL      CHECK ( price <= 5.00 )  
  beer CHAR(20)   CONSTRAINT beer_check  
                  CHECK ( beer IN (SELECT name FROM Beers)),  
);
```

Not possible in Postgre, Oracle
Possible: IN ('Blue', 'Bud')