CS2102: Database Systems

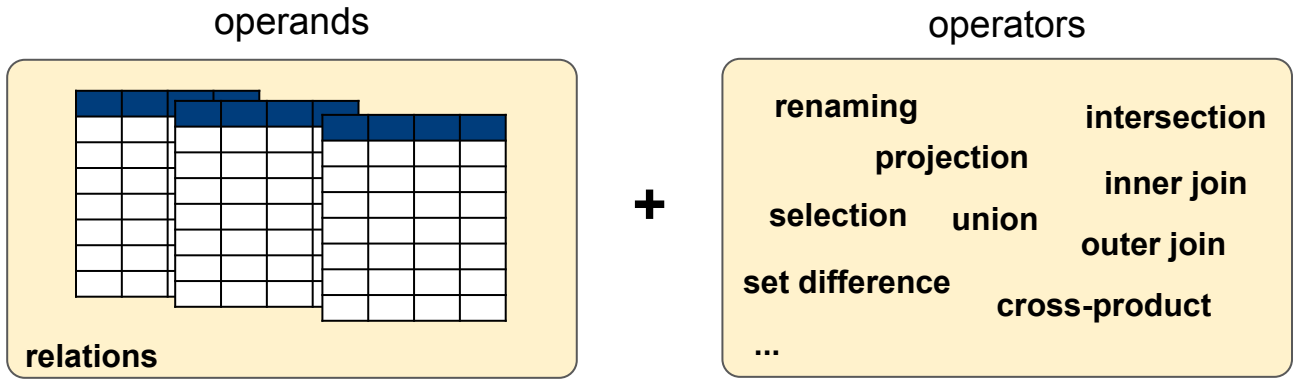
Lecture 3 — SQL (Part 1)

Course Logistics

- Project
  - Team registration will be open soon
  - Check upcoming announcement on LumiNUS with link to submission form
  - Just one member needs to register the team (use your NUSNET IDs for the registrations)

Quick Recap: Relational Algebra

Relational Algebra = operands + operators



- Relation Model & Relational Algebra — theoretical framework to
  - design databases for an RDBMS
  - query data stored in an RDBMS
  - Build applications on top of an RDBMS

Overview

- **SQL — overview**
  - History and usages
  - SQL language groups
- Creating a database with SQL
  - Basic DDL & DML commands
  - Defining integrity constraints
  - Advanced: deferrable constraints
- Modifying a database with SQL
  - Basic DDL commands

SQL — Structured Query Language

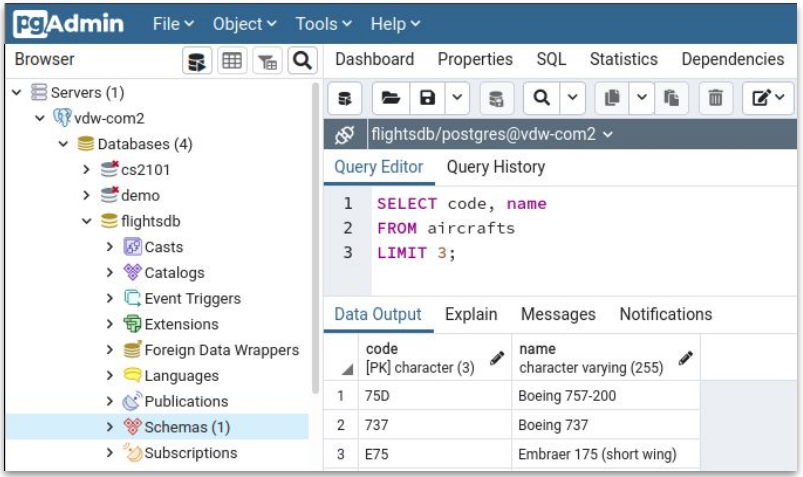
- De-facto standard language to "talk" to RDBMS: **SQL**
  - Developed Donald D. Chamberlin and Raymond F. Boyce (IBM Research, 1974)
  - Originally called SEQUEL (Structured English Query Language)
  - SQL is not a general-purpose language (such as Python, Java, C++, etc.) but a **domain-specific language**
  - SQL is a **declarative language**: focus on *what* to compute, not on *how* to compute
    (Relational Algebra is procedural: complete relational expression have to be specified)

- SQL Standard
  - First standard: SQL-86; most recent standard: SQL-2019 (new standard every ~3-5 years)
  - New standards introduce new language concepts (e.g., support new features of RDBMS)
  - Many RDBMS add the own "flavor" to SQL

Using SQL

- Interactive SQL: directly writing SQL statements to an interface
  - Command line interface
    e.g., PostgreSQL's **psql** [1]
  - Graphical user interface
    e.g., PostgreSQL's **pgAdmin** [2]

```
           List of relations
 Schema |    Name     | Type  |  Owner
--------+-------------+-------+----------
 public | aircrafts   | table | postgres
 public | airports    | table | postgres
 public | countries   | table | postgres
 public | flightcodes | table | postgres
 public | flights     | table | postgres
(5 rows)

flightsdb=# SELECT code, name FROM aircrafts LIMIT 3;
 code |           name
------+--------------------------
 75D  | Boeing 757-200
 737  | Boeing 737
 E75  | Embraer 175 (short wing)
(3 rows)

flightsdb=#
```



[1] https://www.postgresql.org/docs/current/static/app-psql.html
[2] https://www.pgadmin.org/

Using SQL

- Non-interactive
  - SQL statements are included in an application written a host language
  - Two basic approaches to include SQL in host languages: SLI & CLI
- Statement Level Interface (SLI)
  - Application is a mixture of host language statements and SQL statements
  - Examples: Embedded SQL, Dynamic SQL
- Call Level Interface (CLI)
  - Application is completely written in hist language
  - SQL statements are strings passed as arguments to host language procedures or libraries
  - Examples: ODBC (Open DataBase Connectivity), JDBC (Java DataBase Connectivity)

Statement Level Interface (SLI) — Example

```c
int main()
{
    EXEC SQL WHENEVER NOT FOUND DO BREAK;
    EXEC SQL BEGIN DECLARE SECTION;
    char v_code[32], v_name[32];
    EXEC SQL END DECLARE SECTION;

    // Connect to database
    EXEC SQL BEGIN DECLARE SECTION;
    const char *target = "flightsdb@localhost";
    const char *user = "postgres";
    const char *passwd = "[redacted]";
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT TO :target USER :user USING :passwd;

    // Declare cursor
    EXEC SQL DECLARE c CURSOR FOR
    SELECT code, name FROM aircrafts LIMIT 3;

    // Open cursor
    EXEC SQL OPEN c;

    // Loop through cursor and display results
    for(;;) {
        EXEC SQL FETCH NEXT FROM c INTO :v_code, :v_name;
        printf(">>> code: %s, name: %s\n", v_code, v_name);
    }

    // Cleanup (close cursor, commit, disconnect)
    EXEC SQL CLOSE c;
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;

    return 0;
}
```

```bash
#!/bin/bash

# Run ecpg preprocessor to convert C program with embedded SQL statements
# to normal C code; replaces the SQL invocations with special function calls.
ecpg flightsdb.pgc

# Compile generated C code; requires to include all header files the compiler
# needs to understand the special function calls (files come with PostgreSQL).
gcc -g -I/usr/include/postgresql -c flightsdb.c

# Build output to executable file; also needs access to the header files.
gcc -o flightsdb flightsdb.o -L/usr/include/postgresql -lecpg
```

```
>>> code: 75D, name: Boeing 757-200
>>> code: 737, name: Boeing 737
>>> code: E75, name: Embraer 175 (short wing)
```

in this case the SQL is mixed into the code, need to complie properly

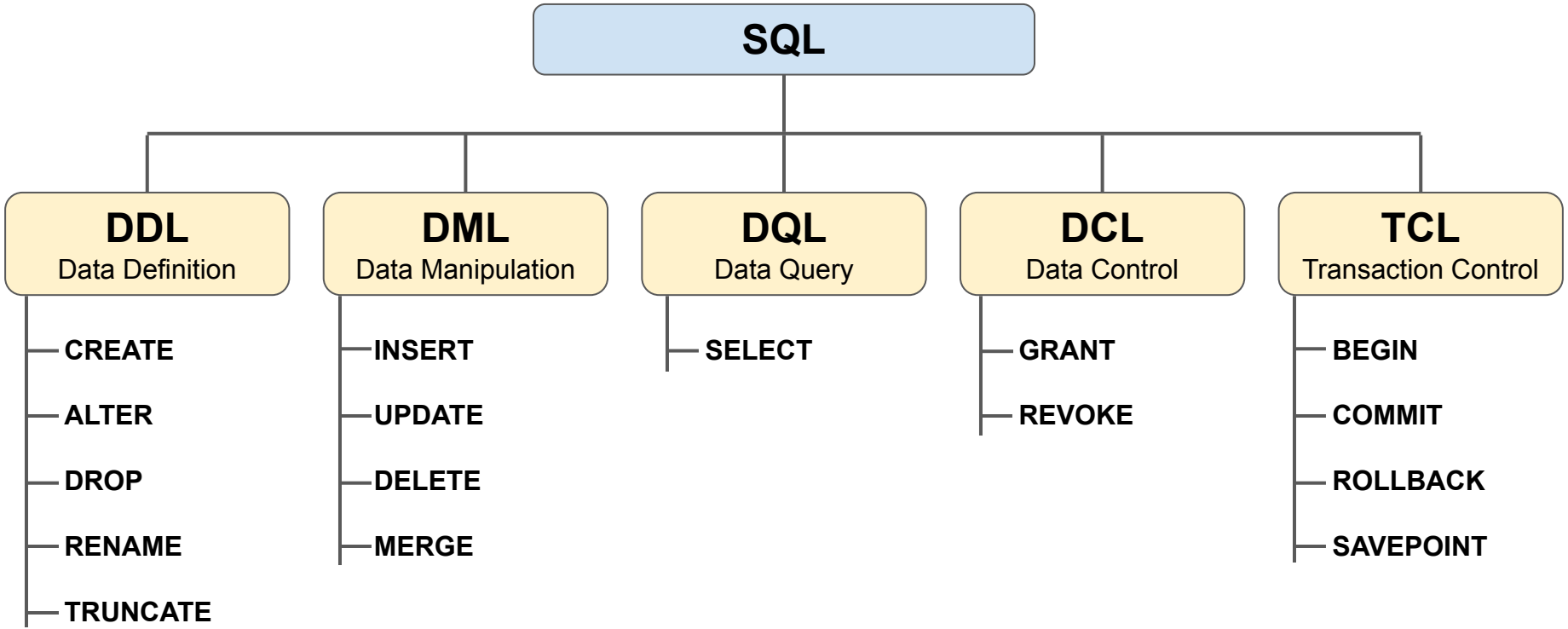
Call Level Interface (CLI) — Example

Call level interface just input SQL as strings into the imported library and its functions

```python
import psycopg2 # Host language library (here psycopg2 for Python)

# Connect to database
db = psycopg2.connect(host="localhost", database="flightsdb", user="postgres", password="[redacted]")

# Create cursor
cursor = db.cursor()

# Open cursor by executing query (string parameter passed to execute() method)
cursor.execute("SELECT code, name FROM aircrafts LIMIT 3")

# Loop over all results until no next tupel is returned
while True:
    row = cursor.fetchone()
    if row is None:
        break
    print(row)

# Cleanup
cursor.close()
db.close()
```

```
('75D', 'Boeing 757-200')
('737', 'Boeing 737')
('E75', 'Embraer 175 (short wing)')
```

SQL — Types of Commands/Statements



Overview

- SQL — overview
  - History and usages
  - SQL language groups
- **Creating a database with SQL**
  - **Basic DDL & DML commands**
  - Defining integrity constraints
  - Advanced: deferrable constraints
- Modifying a database with SQL
  - Basic DDL commands

DDL — Creating Tables

- Basic syntax: definition of table name and attributes (with data types)

Employees (id: **integer**, name: **text**, age: **integer,** role**: text**)

```
CREATE TABLE Employees (
        id          INTEGER,
        name        VARCHAR(50),
        age         INTEGER,
        role        VARCHAR(50)
);
```

- Extended syntax: definition of additional data integrity constraints

Data Types (PostgreSQL)

- Basic data types
  (supported by most RDBMS)

boolean	logical Boolean (true/false)
integer	signed four-byte integer
float8	double precision floating-point number (8 bytes)
numeric [(p,s)]	exact numeric of selectable precision
char(n)	fixed-length character string
varchar(n)	variable-length character string
text	variable-length character string
date	calendar date (year, month, day)
timestamp	date and time

different/many types of strings for different purposes

- Many extended data types
  - Document types: XML, JSON
  - Spatial types: point, line, polygon, circle, box, path
  - Special types: money/currency, MAC/IP address

- Definition user-defined types (UDTs)

Complete list of built-in data types of PostgreSQL: https://www.postgresql.org/docs/current/datatype.html

DML — Inserting Data (Basic Examples)

```
CREATE TABLE Employees (
        id      INTEGER,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50)
);
```

- Example: Inserting 3 employees
  - Specifying all attribute values

    INSERT INTO Employees **VALUES** (101, 'Sarah', 25, 'dev');

  - Specifying selected attribute values

    INSERT INTO Employees (id, name) **VALUES** (102, 'Judy'), (103, 'Max');

Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	null	null
103	Max	null	null

DML — Inserting Data (Basic Examples)

```
CREATE TABLE Employees (
        id      INTEGER,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50) DEFAULT 'sales'
);
```

- Example: Inserting 3 employees
  - Specifying all attribute values

    INSERT INTO Employees **VALUES** (101, 'Sarah', 25, 'dev');

  - Specifying selected attribute values

    INSERT INTO Employees (id, name) **VALUES** (102, 'Judy'), (103, 'Max');

Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	null	**sales**
103	Max	null	**sales**

DML — Deleting Data (Basic Examples)

Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	null	sales
103	Max	null	sales

-- Delete all tuples
DELETE FROM Employees;

Employees

id	name	age	role

table will still exist but will be empty

-- Delete selected tuples
DELETE FROM Employees
WHERE role = 'dev';

can be arbitrarily complex

Employees

id	name	age	role
102	Judy	null	sales
103	Max	null	sales

DML — Updating Data (Basic Examples)

Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	null	sales
103	Max	null	sales

```
-- Sarah's birthday
UPDATE Employees
SET age = age + 1
WHERE name = 'Sarah';
```

Employees

id	name	age	role
101	Sarah	**26**	dev
102	Judy	null	sales
103	Max	null	sales

```
-- New privacy law
UPDATE Employees
SET age = 0;
```

this will change all the null values too

Employees

id	name	age	role
101	Sarah	**0**	dev
102	Judy	**0**	sales
103	Max	**0**	sales

```
-- Uppercasing all strings
UPDATE Employees
SET name = UPPER(name),
    role = UPPER(name);
```

Employees

id	name	age	role
101	**SARAH**	25	**DEV**
102	**JUDY**	null	**SALES**
103	**MAX**	null	**SALES**

Overview

- SQL — overview
  - History and usages
  - SQL language groups
- **Creating a database with SQL**
  - Basic DDL & DML commands
  - **Defining integrity constraints**
  - Advanced: deferrable constraints
- Modifying a database with SQL
  - Basic DDL commands

Prerequisite — Handling *null* Values

- Recall: rules of handling *null* values
  - The result of a comparison operation with *null* is ***unknown***
  - The result of an arithmetic operation with *null* is ***null***

➜ Three-valued logic: true, false, unknown

Assume that value of x is *null*

x < 2020	➜ ***unknown***
x = null	➜ ***unknown***
x <> null	➜ ***unknown***
x + 5	➜ ***null***

- Questions
  - *How to check if a values is equal to null?*
  - *How to treat null values as ordinary values for comparison?*

Important for writing SQL queries & checking integrity constraints!

IS (NOT) NULL Comparison Predicate

- Check if a values is equal to null (since "=" would return unknown)
  - If x is a *null* value ➜ "x **IS NULL**" evaluates to true
  - If x is a non-*null* value ➜ "x **IS NULL**" evaluates to false

  } vice versa for "x **IS NOT NULL**"

- Equivalence
  - "x **IS NOT NULL**" is equivalent to "**NOT** (x **IS NULL**)"

x	y
1	1
1	2
null	1
null	*null*

x	y	x IS NULL	y IS NULL
1	1	***false***	***false***
1	2	***false***	***false***
null	1	***true***	***false***
null	*null*	***true***	***true***

Quick Throwback to Last Lecture

Teams

ename	pname	hours
Sarah	BigAI	10
Sam	BigAI	5
Bill	BigAI	15
Judy	GlobalDB	20
Max	GlobalDB	5
Sarah	GlobalDB	10
Emma	GlobalDB	35
Max	CoreOS	40
Bill	CoreOS	30
Sam	CoolCoin	40
Sarah	CoolCoin	25
Emma	CoolCoin	10

Managers

name	office
Judy	#03-20
Jack	#03-10

How many **rows & columns** has the result of the algebra expression below?

$$\sigma_{\text{ename}=null}(\text{Managers} \bowtie_{\text{name=enam}} \text{Teams})$$

e

$$\sigma_{\text{ename IS NULL}}(\text{Managers} \bowtie_{\text{name=enam}} \text{Teams})$$

e

IS (NOT) NOT DISTINCT Comparison Predicate

- "x **IS DISTINCT FROM** y"

  this will treat null as a value itself instead of a boolean

  - equivalent to "x <> y" if x and y are non-null values
  - if x and y both null ➜ evaluates to ***false***
  - if only one value is null ➜ evaluates to ***true***

  } vice versa for "x **IS NOT DISTINCT FROM** y"

- Equivalence
  - "x **IS NOT DISTINCT FROM** y" is equivalent to "**NOT** (x **IS DISTINCT FROM** y)"

x	y
1	1
1	2
null	1
null	*null*

x	y	x <> y	x IS DISTINCT FROM y
1	1	**FALSE**	**FALSE**
1	2	**TRUE**	**TRUE**
null	1	**null**	**TRUE**
null	*null*	**null**	**FALSE**

DDL — Data Integrity Constraints: Overview

- Types of Constraints ("named" or "unnamed")
  - Not-null constraints
  - Unique constraints
  - Primary key constraints
  - Foreign key constraints
  - General constraints

A constraint is violated if it evaluates to ***false***

- Constraint specifications (difference "where" a constraint is specified)
  - Column constraint: applies to single column, specified at column definition
  - Table constraint: applies to one or more columns, specified after all column definitions
  - Assertion: stand-alone command (**create assertion** ...)

Not-Null Constraints

- Example: the id or name of an employee cannot be *null*

unnamed constraint (name assigned by DBMS)

```
CREATE TABLE Employees (
        id      INTEGER NOT NULL,
        name    VARCHAR(50) NOT NULL,
        age     INTEGER,
        role    VARCHAR(50),
);
```

named constraint (easier bookkeeping)

```
CREATE TABLE Employees (
        id      VARCHAR(50) CONSTRAINT nn_id NOT NULL,
        name    VARCHAR(50) CONSTRAINT nn_name NOT NULL,
        age     INTEGER,
        role    VARCHAR(50),
);
```

- Not-null constraint violation:
  - There exists a tuple t ∈ Employees where "t.id IS NOT NULL" evaluates to ***false***
  - There exists a tuple t ∈ Employees where "t.name IS NOT NULL" evaluates to ***false***

Unique Constraints

- Example: the id of an employee must be unique

unnamed column constraint

```
CREATE TABLE Employees (
        id      INTEGER UNIQUE,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50)
);
```

named column constraint

```
CREATE TABLE Employees (
        id      INTEGER CONSTRAINT u_id UNIQUE,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50)
);
```

unnamed table constraint

```
CREATE TABLE Employees (
        id      INTEGER,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50),
        UNIQUE (id)
);
```

named table constraint

```
CREATE TABLE Employees (
        id      INTEGER,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50),
        CONSTRAINT u_id UNIQUE (id)
);
```

Unique Constraints

- Unique constraint for more than one attribute / column
  - Can only be specified using table constraints
  - Example: Each pair of employee name and project name must be unique

Teams (eid: **integer**, pname: **text**, hours: **integer**)

unnamed table constraint

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER,
        UNIQUE (eid, pname)
);
```

named table constraint

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER,
        CONSTRAINT u_allocation UNIQUE (eid, pname)
);
```

Unique Constraints

Quick Quiz: Is the unique constraint of table "Teams" violated in the example below?

Solution
- NO, based on the definition below

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER,
        UNIQUE (eid, pname)
);
```

Teams

eid	pname	hours
101	BigAI	10
105	BigAI	5
102	GlobalDB	20
101	null	null
101	null	null
103	CoreOS	40
109	CoreOS	null

false unknown unknown

- Unique constraint violation
  - For any two tuples t_i, $t_k \in$ Teams:
  - "(t_i.eid <> t_k.eid) or (t_i.pname) <> t_k.pname)" evaluates to ***false***

Primary Key Constraints

Quick Quiz: What is the difference between using "primary key" and "unique not null"?

Solution

- Only 1 "primary key" constraint possible but multiple "unique not null" constraints
- Foreign key constraints only applicable to primary keys in referenced table

- Quick recap: primary key
  - Selected key uniquely identifying tuples in a table
  - Prime attributes (i.e. attributes of primary key) cannot be null

Employees (id: **integer**, name: **text**, age: **integer,** role**: text**)

```
CREATE TABLE Employees (
        id      INTEGER PRIMARY KEY,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50)
);
```

```
CREATE TABLE Employees (
        id      INTEGER UNIQUE NOT NULL,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50)
);
```

same effect

Primary Key Constraints

- Primary key constraint for more than one attribute / column

Teams (eid: **integer**, pname: **text**, hours: **integer**)

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER,
        PRIMARY KEY (eid, pname)
);
```

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER,
        CONSTRAINT pk_allocation PRIMARY KEY (eid, pname)
);
```

Sidenote

- Specification of constraints — basic rules
  - All constraints can be specified "named" or "unnamed"
    (unnamed constraints still get named by the DBMS in a meaningful way; names can be looked up)
  - All column constraints can be specified as table constraints
    (exception: "not null" only possible as column constraint)
  - Table columns referring to a single column can be specified as column constraint
  - Column and table constraints can be combined (even w.r.t. to the same column)

```
CREATE TABLE Employees (
        id      INTEGER NOT NULL,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50),
        UNIQUE (id)
);
```

Foreign Key Constraints

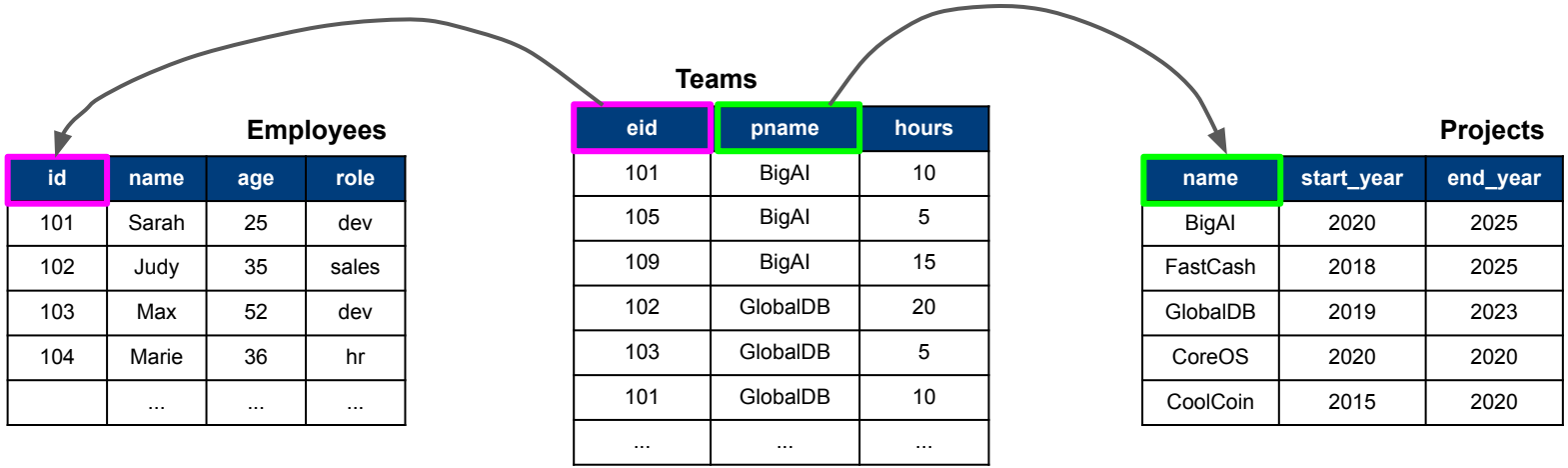
- Quick recap: foreign key constraint
  - Subset of attributes of relation A if it refers to the primary key in a relation B

Employees (id: **integer**, name: **text**, age: **integer,** role**: text**, hired: **date**)

Teams (eid: **integer**, pname: **text**, hours: **integer**)

Projects (name: **text**, start_year: **integer**, end_year: **integer**)

Foreign key constraints
Teams.eid → Employees.id
Teams.pname → Projects.name



Employees

id	name	age	role
101	Sarah	25	dev
102	Judy	35	sales
103	Max	52	dev
104	Marie	36	hr
	...	...	...

Teams

eid	pname	hours
101	BigAI	10
105	BigAI	5
109	BigAI	15
102	GlobalDB	20
103	GlobalDB	5
101	GlobalDB	10
...	...	...

Projects

name	start_year	end_year
BigAI	2020	2025
FastCash	2018	2025
GlobalDB	2019	2023
CoreOS	2020	2020
CoolCoin	2015	2020

Foreign Key Constraints

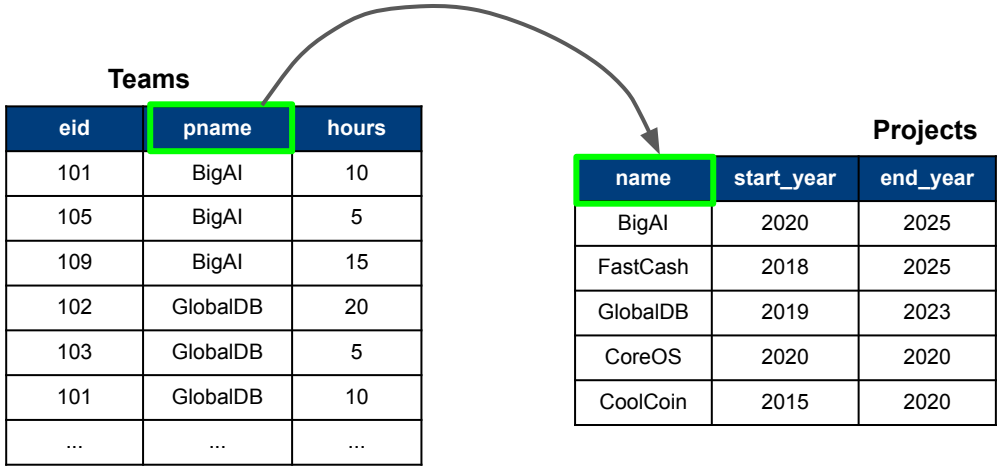
```
CREATE TABLE Employees (
        id      INTEGER PRIMARY KEY,
        name    VARCHAR(50),
        age     INTEGER,
        role    VARCHAR(50)
);
```

```
CREATE TABLE Projects (
        name        VARCHAR(50) PRIMARY KEY,
        start_year  INTEGER,
        end_year    INTEGER
);
```

```
CREATE TABLE Teams (
        eid         INTEGER,
        pname       VARCHAR(100),
        hours       INTEGER,
        PRIMARY KEY (ename, pname),
        FOREIGN KEY (eid) REFERENCES Employees (id),
        FOREIGN KEY (pname) REFERENCES Projects (name)
);
```

Foreign Key Constraints — Violations

- Quick recap: each foreign key in referencing relation must
  - appear as primary key in referenced relation OR
  - be a *null* value



Teams

eid	pname	hours
101	BigAI	10
105	BigAI	5
109	BigAI	15
102	GlobalDB	20
103	GlobalDB	5
101	GlobalDB	10
...	...	...

Projects

name	start_year	end_year
BigAI	2020	2025
FastCash	2018	2025
GlobalDB	2019	2023
CoreOS	2020	2020
CoolCoin	2015	2020

Note: Trying to insert or update a tuple in "Teams" with a new project name that is not in "Project" will always violate the foreign constraint.

Questions:

- What happens if the first tuple in "Project" should be deleted?
- What if the project "BigAI" should be renamed to "SmartAI"?

Solution

- By default: error

Foreign Key Constraints — Violations

- Extend syntax to specify behavior when data in referenced table changes
  - Specify action in case of the violation of a foreign key constraint
  - **ON DELETE/UPDATE <action>** to distinguish action w.r.t. to a delete of update in referenced table
  - Both specifications are optional

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER,
        PRIMARY KEY (ename, pname),
        FOREIGN KEY (eid) REFERENCES Employees (id) ON DELETE <action> ON UPDATE <action>,
        FOREIGN KEY (pname) REFERENCES Projects (name) ON DELETE <action> ON UPDATE <action>
);
```

Foreign Key Constraints — Violations

- Possible actions for **on delete** and **on update**

NO ACTION	rejects delete/update if it violates constraint (default value)
RESTRICT	similar to "no action" except that check of constraint cannot be deferred (deferrable constraints are discussed in a bit)
CASCADE	propagates delete/update to referencing tuples
SET DEFAULT	updates foreign keys of referencing tuples to some default value (important: default value must be a primary key in the referenced table!)
SET NULL	updates foreign keys of referencing tuples to *null* (important: corresponding column must allowed to contain *null* values!)

Foreign Key Constraints

Quick Quiz: The SQL command below is correct but what will cause problems. Why?

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER,
        PRIMARY KEY (eid, pname),
        FOREIGN KEY (eid) REFERENCES Employees (id) ON DELETE NO ACTION ON UPDATE CASCADE,
        FOREIGN KEY (pname) REFERENCES Projects (name) ON DELETE SET NULL ON UPDATE CASCADE
);
```

optional since it is the default action

- Effects on handling violations of foreign key constraints
  - Updates of "Employees.id" and "Projects.name" are propagated to affected tuples in "Teams"
  - Deleting a project will set "Teams.pname" to *null* for employees working on that project
  - Deleting an employee will raise an error if that employee is still assigned to a team

Quick Quiz

Solution

- "FOREIGN KEY (pname) REFERENCES Projects (name) ON DELETE SET NULL" will try to set Teams.pname to NULL if a corresponding project would get deleted
- Problem: "pname" is a prime attribute, i.e., part of the primary key, and prime attributes are not allowed to be NULL
- Violation of primary key constraint ➜ error!

Foreign Key Constraints — Example

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER,
        PRIMARY KEY (eid, pname),
        FOREIGN KEY (eid) REFERENCES Employees (id) ON UPDATE CASCADE,
        FOREIGN KEY (pname) REFERENCES Projects (name) ON UPDATE CASCADE
);
```

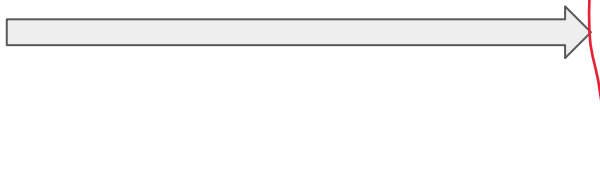
Projects

name	start_year	end_year
BigAI	2020	2025
FastCash	2018	2025
...	...	...

Teams

eid	pname	hours
101	BigAI	10
105	BigAI	5
109	BigAI	15
102	GlobalDB	20
...	...	...

```
UPDATE Projects
SET name = 'SmartAI'
WHERE name = 'BigAI';
```



Projects

name	start_year	end_year
SmartAI	2020	2025
FastCash	2018	2025
...	...	...

Teams

eid	pname	hours
101	SmartAI	10
105	SmartAI	5
109	SmartAI	15
102	GlobalDB	20
...	...	...

Foreign Key Constraints — Example

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100) DEFAULT 'FastCash',        -- default value must be primary key in "Projects"!
        hours           INTEGER,
        PRIMARY KEY (eid, pname),
        FOREIGN KEY (eid) REFERENCES Employees (id) ON UPDATE CASCADE,
        FOREIGN KEY (pname) REFERENCES Projects (name) ON UPDATE CASCADE ON DELETE SET DEFAULT
);
```

Projects

name	start_year	end_year
BigAI	2020	2025
FastCash	2018	2025
...	...	...

Teams

eid	pname	hours
101	BigAI	10
105	BigAI	5
109	BigAI	15
102	GlobalDB	20
...	...	...

```
DELETE FROM Projects
WHERE name = 'BigAI';
```

Projects

name	start_year	end_year
FastCash	2018	2025
...	...	...

Teams

eid	pname	hours
101	FastCash	10
105	FastCash	5
109	FastCash	15
102	GlobalDB	20
...	...	...

Foreign Key Constraints

- Practical considerations
  - Specified constraints might not behave as expected (e.g., **SET NULL** issue with prime attributes)
  - Particularly **ON DELETE CASCADE** can have very bad consequences
  - **CASCADE** may significantly affect overall performance

➜ **Careful design and specification of foreign key constraints is crucial!**

Check Constraints

- **CHECK** constraint
  - Most basic general constraint (i.e., not a structural integrity constraint)
  - Allows to specify that column values must satisfy a Boolean expression
  - Scope: one table, single row

- Example: The hours an employee is allocated to a project must be > 0

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER check (hours > 0),                                  unnamed version
        -- hours        INTEGER constraint positive_hours check (hours > 0),       named version
        PRIMARY KEY (eid, pname),
        FOREIGN KEY (eid) REFERENCES Employees (id),
        FOREIGN KEY (pname) REFERENCES Projects (name)
);
```

Check Constraints

- **CHECK** constraints can refer to multiple columns
  - Example: The start year of a project cannot be larger value than the end year

```
CREATE TABLE Projects (
        name            VARCHAR(50) PRIMARY KEY,
        start_year      INTEGER,
        end_year        INTEGER,
        -- CHECK (start_year <= end_year),
        CONSTRAINT valid_lifetime CHECK (start_year <= end_year)
);
```

Check Constraints

- **CHECK** constraints can be arbitrarily complex Boolean expressions
  - Example: minimum hour requirements for different projects

```
CREATE TABLE Teams (
        eid             INTEGER,
        pname           VARCHAR(100),
        hours           INTEGER,
        PRIMARY KEY (eid, pname),
        FOREIGN KEY (eid) REFERENCES Employees (id),
        FOREIGN KEY (pname) REFERENCES Projects (name),
        CHECK (
                (pname = 'CoreOS' AND hours >= 30)
                OR
                (pname <> 'CoreOS' AND hours > 0)
        )
);
```

Assertions

- **CREATE ASSERTION** statement (since SQL-92)
  - Formulation of (almost) arbitrary constraints
  - Scope: multiple tables, multiple rows
  - Example: "Each project must have at least one team member being 30 or older"

- Assertion in practice: various potential side effects and limitations, e.g.:
  - Assertions cannot modify the data
  - No proper error handling
  - Not linked to a specific table
    (e.g., dropping a table does not affect assertion)

➜ Most RDBMS do not support assertions but **triggers** (more powerful alternative)

Overview

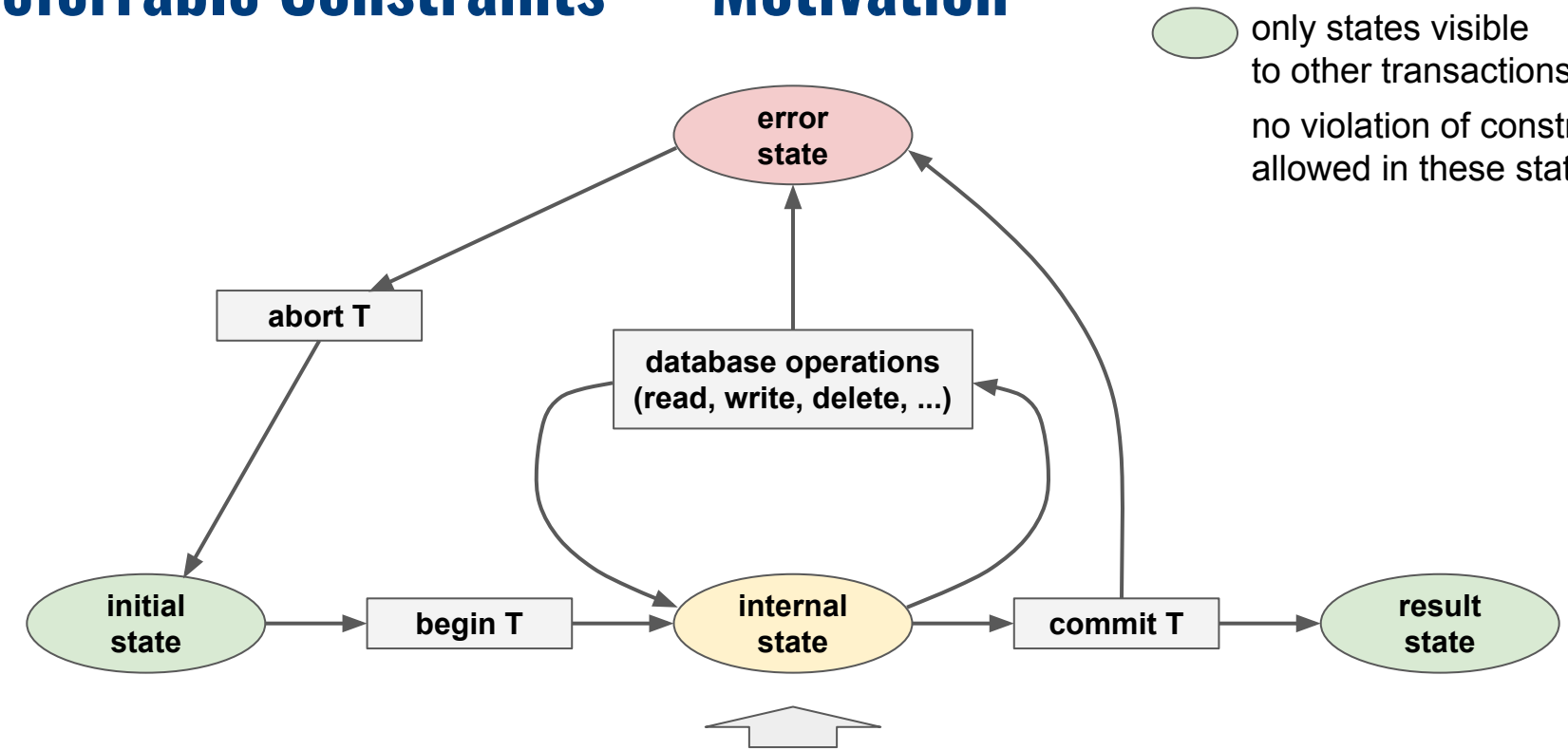
- SQL — overview
  - History and usages
  - SQL language groups
- **Creating a database with SQL**
  - Basic DDL & DML commands
  - Defining integrity constraints
  - **Advanced: deferrable constraints**
- Modifying a database with SQL
  - Basic DDL commands

Deferrable Constraints — Motivation

- Default behavior for constraints
  - Constraints are checked immediately at the end of SQL statement execution
    (even within a transaction containing multiple SQL statements)
  - A violation will cause the statement to be rolled back

- Relaxed constraint checks: **Deferrable Constraints**
  - Check can be deferred for some constraints to the end of a transaction
  - Available for: UNIQUE, PRIMARY KEY, FOREIGN KEY

Deferrable Constraints — Motivation



Deferrable constraints may (temporarily) be violated within the scope of a transaction

Deferrable Constraints — Example

- Motivating example without deferrable constraints

Employees

id	name	manager
101	Sarah	null
102	Judy	101
103	Max	102

```
CREATE TABLE Employees (
        id              INTEGER PRIMARY KEY,
        name            VARCHAR(50),
        manager         INTEGER,
        CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES Employees (id)
                NOT DEFERRABLE -- default value (optional), check if constraint is immediate and cannot be changed
);

INSERT INTO Employees VALUES (101, 'Sarah', null), (102, 'Judy', 101), (103, 'Max', 102);


BEGIN;
DELETE FROM Employees WHERE id = 102;                   -- Judy got fired     ➜ constraint violated ➜ ABORT
UPDATE Employees SET manager = 101 WHERE id = 103;   -- Max gets a new manager
COMMIT;
```

Deferrable Constraints — Example

Employees

id	name	manager
101	Sarah	null
102	Judy	101
103	Max	102

```
CREATE TABLE Employees (
        id              INTEGER PRIMARY KEY,
        name            VARCHAR(50),
        manager         INTEGER,
        CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES Employees (id)
                DEFERRABLE INITIALLY DEFERRED -- check of constraint deferred by default
);

INSERT INTO Employees VALUES (101, 'Sarah', null), (102, 'Judy', 101), (103, 'Max', 102);


BEGIN;
DELETE FROM Employees WHERE id = 102;                -- Judy got fired ➜ constraint violated but not checked
UPDATE Employees SET manager = 101 WHERE id = 103;   -- Max gets a new manager ➜ constraint re-established
COMMIT;
```

Deferrable Constraints — Example

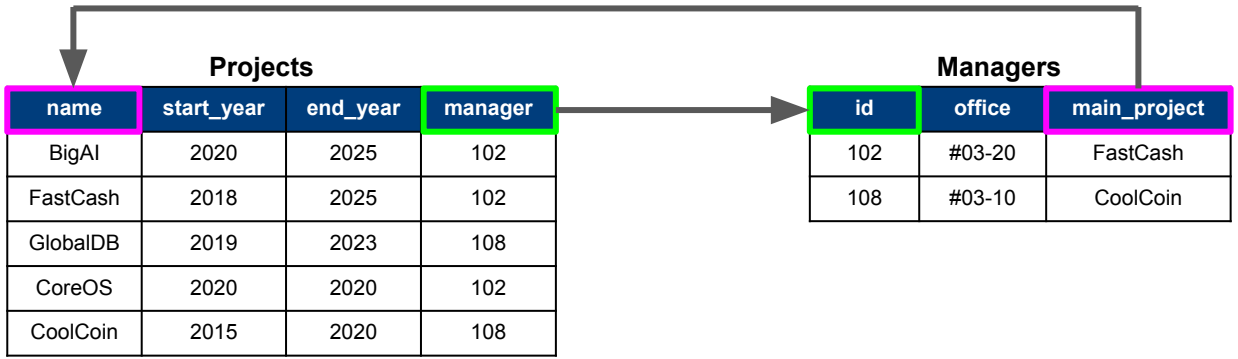
Employees

id	name	manager
101	Sarah	null
102	Judy	101
103	Max	102

```
CREATE TABLE Employees (
        id              INTEGER PRIMARY KEY,
        name            VARCHAR(50),
        manager         INTEGER,
        CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES Employees (id)
                DEFERRABLE INITIALLY IMMEDIATE -- check of constraint immediate by default, but can be changed
);

INSERT INTO Employees VALUES (101, 'Sarah', null), (102, 'Judy', 101), (103, 'Max', 102);


BEGIN;
SET CONSTRAINT manager_fkey DEFERRED;          -- Set check of constraint from "immediate" to "deferred"
DELETE FROM Employees WHERE id = 102;          -- Judy got fired ➜ constraint violated buy not checked
UPDATE Employees SET manager = 101 WHERE id = 103;   -- Max gets a new manager ➜ constraint re-established
COMMIT;                                        this will changed everything back to original again
```

Deferrable Constraints — Benefits

- No need to care about order of SQL statements within a transaction
- Allows for cyclic foreign key constraints



Projects

name	start_year	end_year	manager
BigAI	2020	2025	102
FastCash	2018	2025	102
GlobalDB	2019	2023	108
CoreOS	2020	2020	102
CoolCoin	2015	2020	108

Managers

id	office	main_project
102	#03-20	FastCash
108	#03-10	CoolCoin

- Performance boost when constraint checks are bottleneck
  - Example: batch insert of large number of tuples

Deferrable Constraints — (Potential) Downsides

- Troubleshooting can be more difficult
- Data definition no longer unambiguous
- Performance penalty when performing queries

Overview

- SQL — overview
  - History and usages
  - SQL language groups
- Creating a database with SQL
  - Basic DDL & DML commands
  - Defining integrity constraints
  - Advanced: deferrable constraints
- **Modifying a database with SQL**
  - Basic DDL commands

DDL — Modifying a Schema

- **ALTER TABLE** statements to modify an existing data definition
  - CREATE TABLE statements do not have to be final data definition
  - Common: adding/dropping column, adding dropping constraints, changing data types

- Examples: Change specification of a single column

```
ALTER TABLE Projects ALTER COLUMN name TYPE VARCHAR(200);      -- change data type to VARCHAR(200)
ALTER TABLE Projects ALTER COLUMN start_year SET DEFAULT 2021; -- set default value of column "start_year"
ALTER TABLE Projects ALTER COLUMN start_year DROP DEFAULT;     -- drop default value of column "start_year"
```

DDL — Modifying a Schema

- Examples: Adding and dropping columns

```
ALTER TABLE Projects ADD COLUMN budget NUMERIC DEFAULT 0.0;   -- add new column with a default value

ALTER TABLE Projects DROP COLUMN budget;                      -- drop column from table
```

cannot drop primary key

- Examples: Adding and dropping constraints

```
ALTER TABLE Teams ADD CONSTRAINT eid_fkey FOREIGN KEY (eid) REFERENCES Employees (id);
        -- add foreign key constraint

ALTER TABLE Teams DROP CONSTRAINT eid_fkey;
        -- drop foreign key constraint (name of constraint might be retrieved from metadata)
```

Reference: https://www.postgresql.org/docs/current/sql-altertable.html

DDL — Drop Tables

- DROP TABLE to delete tables from database
  - Without dependent objects (incl. foreign key constraints, views, etc.)

```
DROP TABLE Projects;

DROP TABLE IF EXISTS Projects;    -- check first if table exists; avoids throwing an error
```

OR can also do something like : CREATE TABLE IF NOT EXISTS

  - With dependent objects (assume foreign key constraint Teams.pname→Projects.name)

```
DROP TABLE Projects;              -- will throw an error because of foreign key constraint

DROP TABLE Projects CASCADE;      -- will delete table "Projects" and foreign key constraint
                                  -- (will not delete table "Teams"!)
```

Summary

- SQL — *the* standard language for RDBMS
  - Different language groups: DDL, DML, SQL, DCL, TCL
- Focus in this lecture: DDL and DML
  - DDL: **CREATE TABLE**, **ALTER TABLE**, **DROP TABLE**
  - DML: **INSERT**, **UPDATE**, **DELETE**
- Key challenge: specification of integrity constraints
  - **NOT NULL**, **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY**, **CHECK**
  - Specification actions in case of foreign key constraint violations (**ON UPDATE/DELETE**)
  - Relaxed checks of violations with deferrable constraints