Introduction To Database Design

Demitri Muna

Ohio State University

SciCoder 8: Yale

3 August 2016

Why Use A Database?

- Astronomy today generates more data each year than every year before it – combined.
- OK, I completely made that up, but enormous amounts of data are now generated (may be true with LSST!).
- Searching for what you are looking for in hundreds or thousands of FITS files is becoming increasingly unwieldy, repetitive, and time consuming.
- You are effectively writing your own search engine – let the CS community do that for you. (They're better than you at it anyway.)

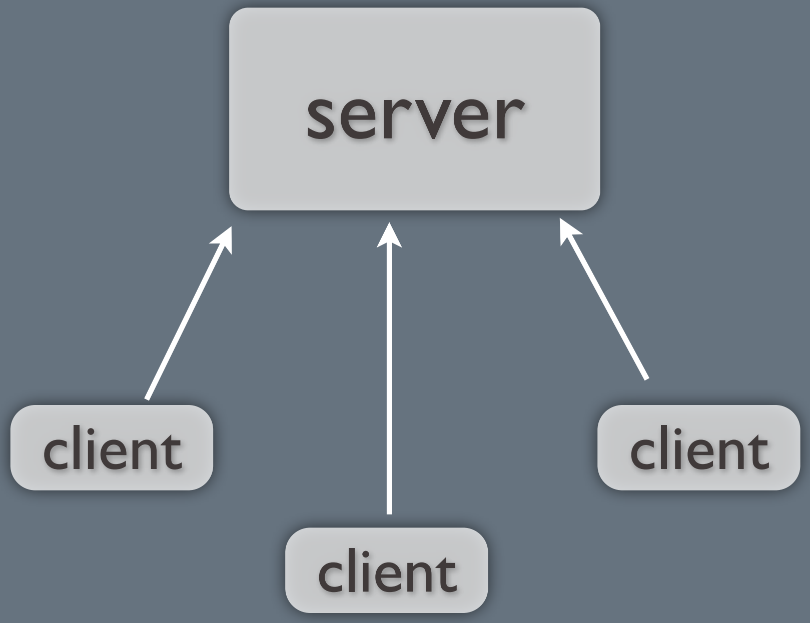
Benefits of a Database

- One place to store all of your data, without worrying about file formats.
- Add other people's data for cross referencing/correlation to the same pool with little effort
- Searches are fast, whether you have one hundred objects or millions of them. The time taken for a full search does not scale geometrically as it would with simple files.
- You don't have to write a new program when you change the criteria for your search.
- The database language (SQL) is very easy to learn, and even easier to read.

Kinds of Databases

There are (mainly) three types of databases.

Client/Server SQL Database



- server process
- multiple, simultaneous clients
- handles concurrency (i.e. can handle multiple people/processes trying to write to the same database)
- can write custom functions in several languages

Examples: PostgreSQL, Oracle, MySQL

Kinds of Databases

There are (mainly) three types of databases.

File-Based Database

- no need to install a separate running server
- the whole database is a single file
- database file is cross platform (email the database to anyone!)
- fully open source
- basically a C library – can embed a full SQL database in your own program
- can be used as a file format – search engine for free
- hooks from just about any programming language

file on disk

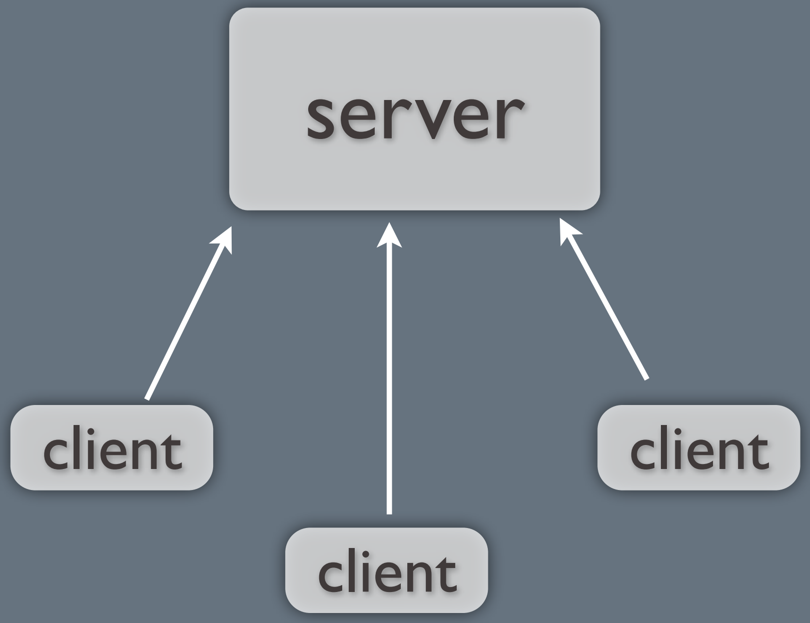
Examples: SQLite

Kinds of Databases

There are (mainly) three types of databases.

NoSQL Database

- “document” based storage
- no need to know structure before using
- essentially key-value based storage
- can be distributed
- scalable to large sizes
- queries can be more difficult to write
- most data fields don’t need to be searched



Examples: MongoDB, CouchDB

Which Database Do I Use?

I recommend one of two, depending on your needs:

PostgreSQL

- need many people to access at once
- very large size (TB+)
- back end for a web site
- need high-performance
- need multi-threading (i.e. use many CPU cores)

SQLite

- need a search engine for yourself
- need a database embedded in your program
- want to easily sent a complex data set to another person

If someone suggests using Oracle, punch them.

Which Database Do I Use?

Why not NoSQL?

- astronomical data is *highly* structured; most of the benefits of NoSQL are through the handling of unstructured data
- extremely complex queries are (relatively) easy using SQL; they are difficult (or impossible) with NoSQL
- the majority of astronomical data should be easily queryable

Designing a Database

- You can take several full courses in database design and optimization, but for our needs, the basics are actually pretty straightforward (although maybe not immediately intuitive).
- The main principle is normalization, which is the idea that no information in the database is duplicated. You will frequently have to reorganize your data to accomplish this.
- The design (or blueprint) of the database is called a schema.

Creating a Database

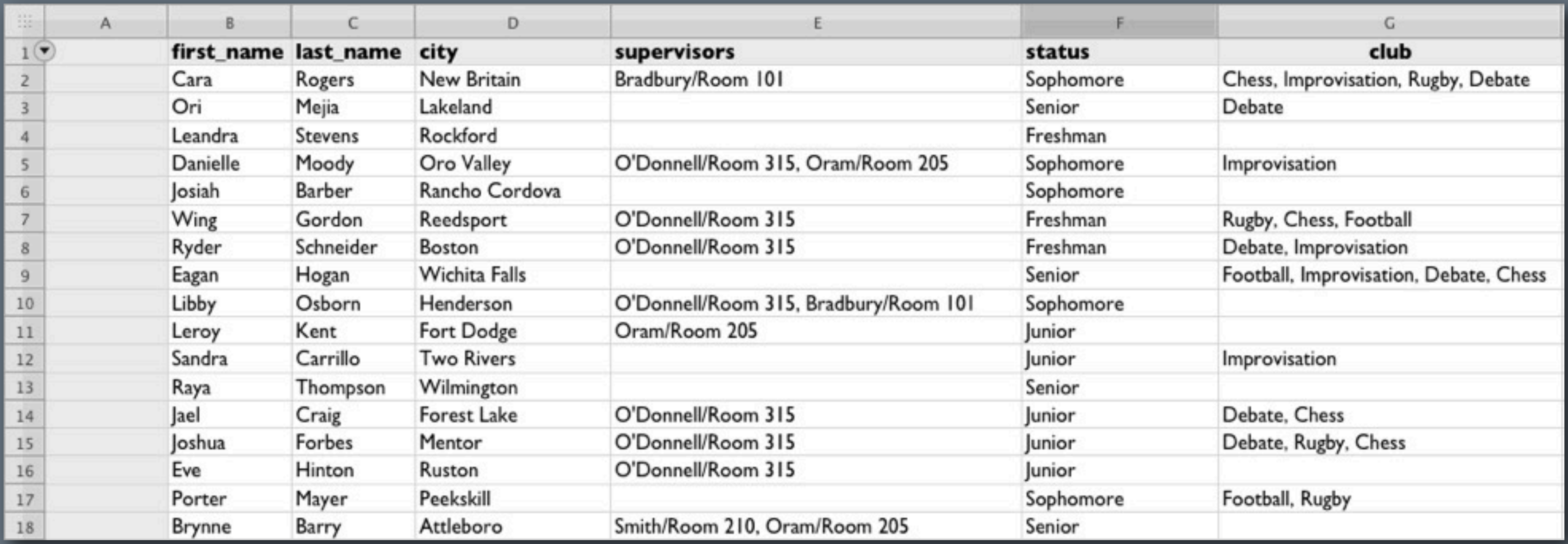
The typical work flow to create a database:

- Design a schema. Plan for future expansion/possibilities.
- Write a script to convert the data in its given form (e.g. ASCII files, FITS files) into the new normalized form.
- Import the data into the database.

Designing A Schema

The best way to illustrate this is through a basic example.
What are the problems with data in this form?

see file "student_data.txt"



	A	B	C	D	E	F	G
1		**first_name**	**last_name**	**city**	**supervisors**	**status**	**club**
2		Cara	Rogers	New Britain	Bradbury/Room 101	Sophomore	Chess, Improvisation, Rugby, Debate
3		Ori	Mejia	Lakeland		Senior	Debate
4		Leandra	Stevens	Rockford		Freshman	
5		Danielle	Moody	Oro Valley	O'Donnell/Room 315, Oram/Room 205	Sophomore	Improvisation
6		Josiah	Barber	Rancho Cordova		Sophomore	
7		Wing	Gordon	Reedsport	O'Donnell/Room 315	Freshman	Rugby, Chess, Football
8		Ryder	Schneider	Boston	O'Donnell/Room 315	Freshman	Debate, Improvisation
9		Eagan	Hogan	Wichita Falls		Senior	Football, Improvisation, Debate, Chess
10		Libby	Osborn	Henderson	O'Donnell/Room 315, Bradbury/Room 101	Sophomore	
11		Leroy	Kent	Fort Dodge	Oram/Room 205	Junior	
12		Sandra	Carrillo	Two Rivers		Junior	Improvisation
13		Raya	Thompson	Wilmington		Senior	
14		Jael	Craig	Forest Lake	O'Donnell/Room 315	Junior	Debate, Chess
15		Joshua	Forbes	Mentor	O'Donnell/Room 315	Junior	Debate, Rugby, Chess
16		Eve	Hinton	Ruston	O'Donnell/Room 315	Junior	
17		Porter	Mayer	Peekskill		Sophomore	Football, Rugby
18		Brynne	Barry	Attleboro	Smith/Room 210, Oram/Room 205	Senior	

Designing A Schema

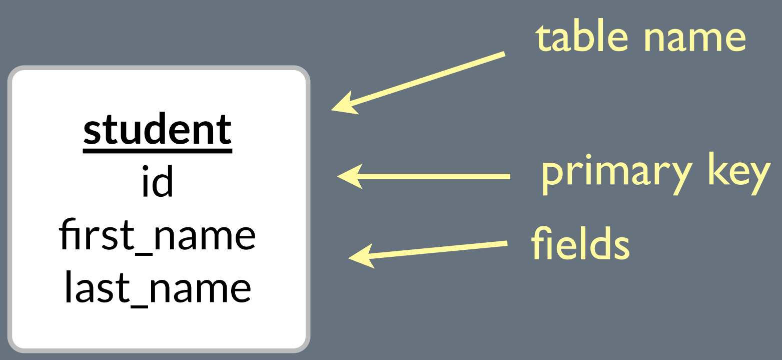
- Data is frequently repeated. A misspelling can lead to lost (or at least orphaned) data.
- Repeated data consumes more disk space unnecessarily.
- A spreadsheet doesn’t handle several pieces of data related to a single object.
- Only simple reports or queries are easy to make.
- You can’t put gigabytes of data into a spreadsheet.

Caveat

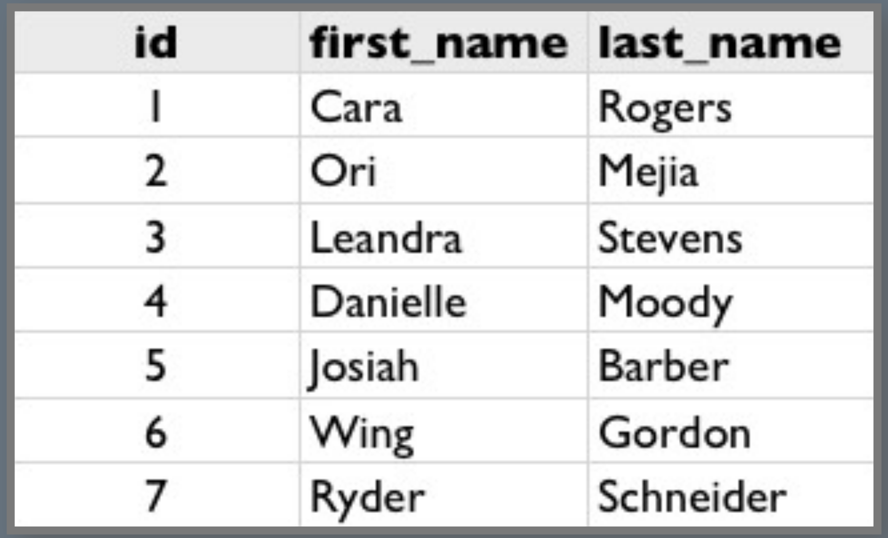
The example presented is a toy model. Inefficiencies here might seem trivial, but in a real dataset will quickly scale, e.g. wasting disk space, hinder efficient searches, etc.

Factoring the Data

Create a table for each "object" in your data model.
Think of a table as a single spreadsheet.



Student
table sample

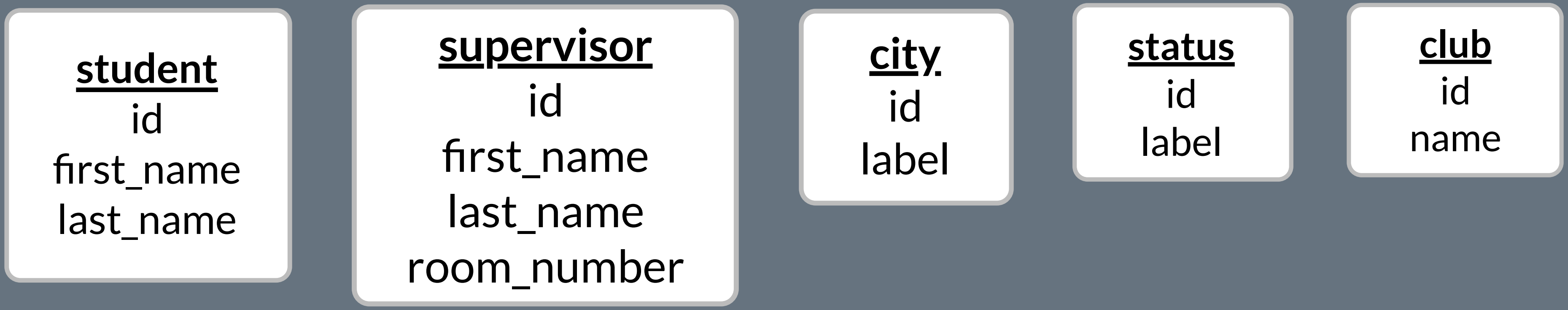


id	first_name	last_name
1	Cara	Rogers
2	Ori	Mejia
3	Leandra	Stevens
4	Danielle	Moody
5	Josiah	Barber
6	Wing	Gordon
7	Ryder	Schneider

Each table must have a primary key, which is a value that uniquely identifies a row. Typically this value is an integer.

It should not be meaningful or linked to any value in the table.
Repeat for every "noun" in the data model, e.g. 'city', 'status'.

Factoring the Data



student
id
first_name
last_name

supervisor
id
first_name
last_name
room_number

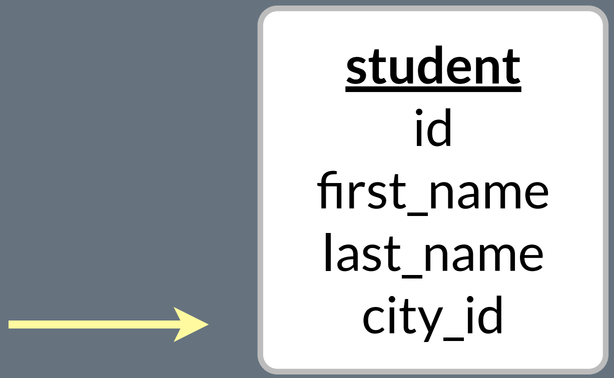
city
id
label

status
id
label

club
id
name

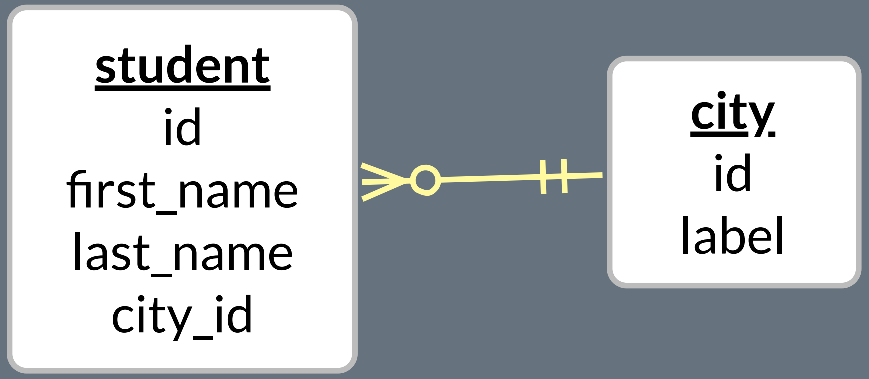
How do we know which city a student is from? Which clubs they belong to? Which supervisors they have?

Let’s start with the city. To identify a student with a city, we add a new field:

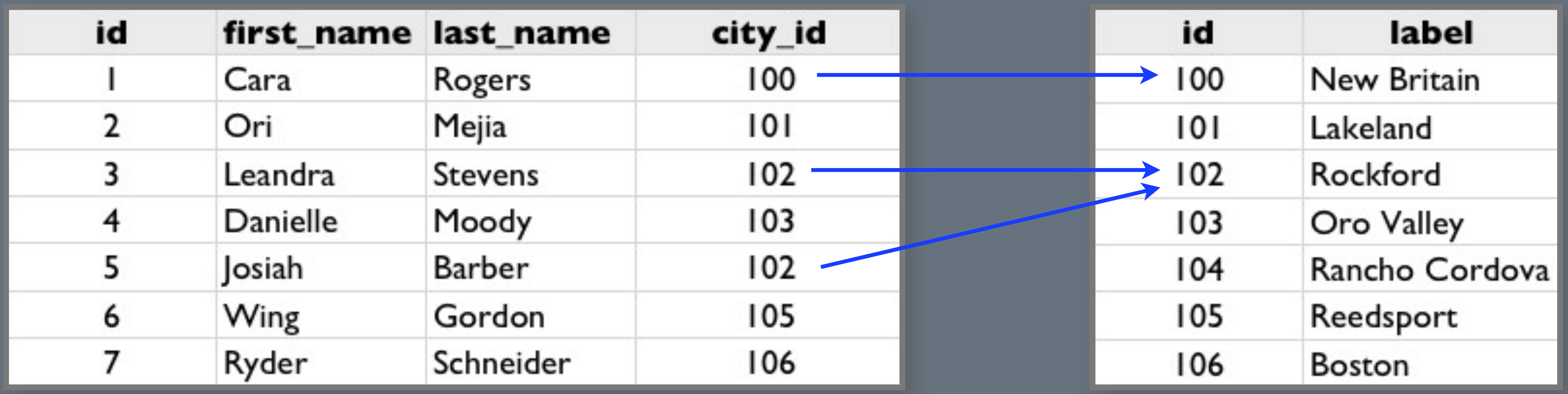


student
id
first_name
last_name
city_id

One-to-Many Relationship



- The field `city_id` in `student` maps to the primary key in the table `city` – this is called a *foreign key*.
- This defines a *one-to-many relationship* – one student comes from one city, but one city can have many students.
- In this case, `city` is often called a *lookup table*, as the `city_id` field is used to look up the name of the city.
- An advantage is that the city name is itself is located in one and only one place in the database.



Student table

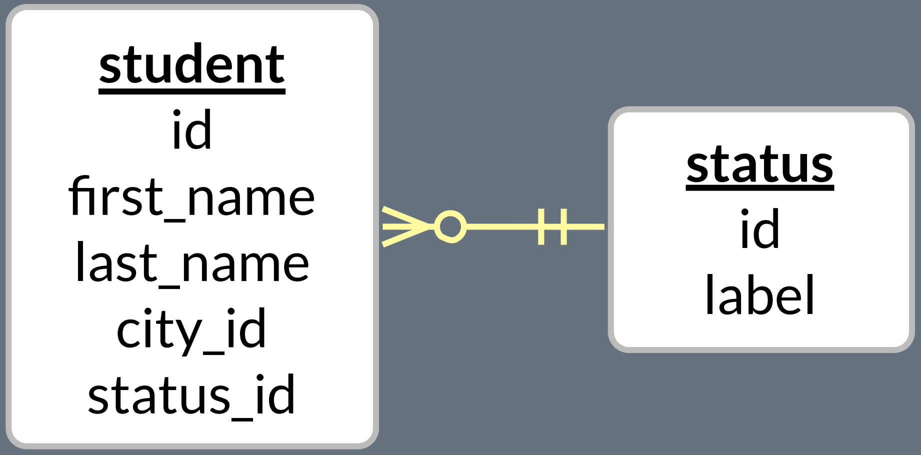
id	first_name	last_name	city_id
1	Cara	Rogers	100
2	Ori	Mejia	101
3	Leandra	Stevens	102
4	Danielle	Moody	103
5	Josiah	Barber	102
6	Wing	Gordon	105
7	Ryder	Schneider	106

City table

id	label
100	New Britain
101	Lakeland
102	Rockford
103	Oro Valley
104	Rancho Cordova
105	Reedsport
106	Boston

Design A Schema

What about the relationship to status? The question to ask is, can a student (ever) have more than one status at a time?



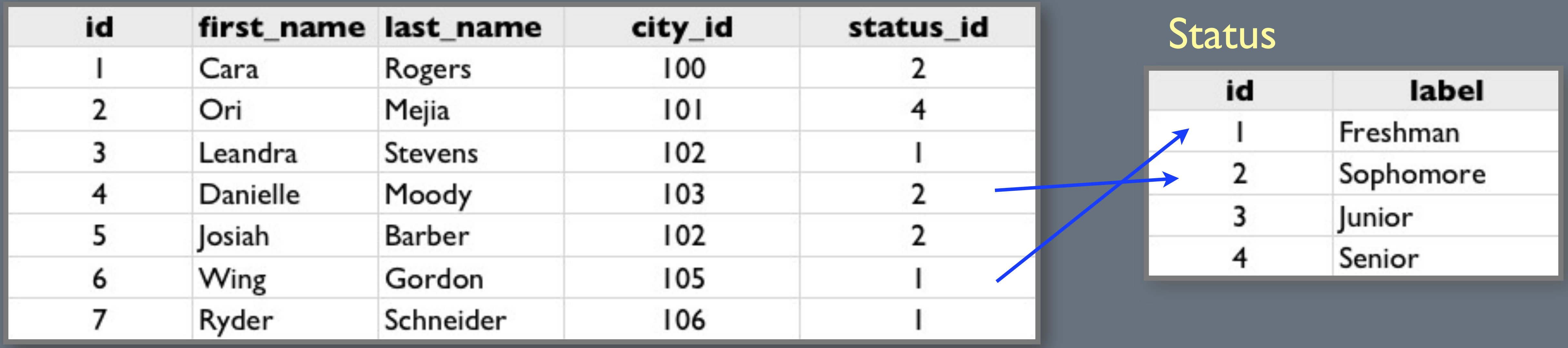
student
id
first_name
last_name
city_id
status_id

status
id
label

This is another one-to-many relationship; `status` is another lookup table.

Student

Status



id	first_name	last_name	city_id	status_id
1	Cara	Rogers	100	2
2	Ori	Mejia	101	4
3	Leandra	Stevens	102	1
4	Danielle	Moody	103	2
5	Josiah	Barber	102	2
6	Wing	Gordon	105	1
7	Ryder	Schneider	106	1

id	label
1	Freshman
2	Sophomore
3	Junior
4	Senior

Many-to-Many Relationship

The relationship to supervisor is trickier as a student can have more than one supervisor, and a supervisor can certainly have more than one student. The same solution doesn’t work:

student
id
first_name
last_name
city_id
status_id
supervisor_id

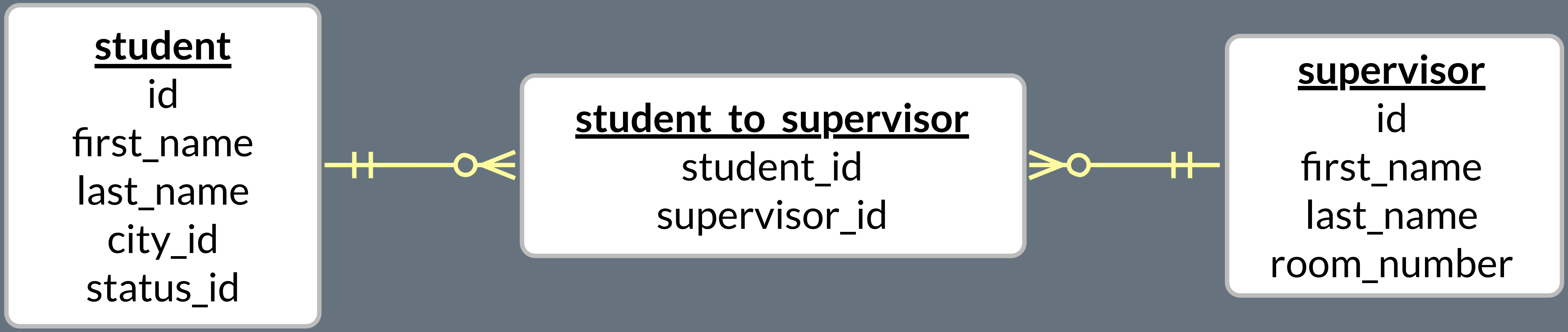
This allows only one supervisor at a time.

student
id
first_name
last_name
city_id
status_id
supervisor_id1
supervisor_id2

- This doesn’t allow more than two supervisors (could happen).
- Space for two supervisor foreign keys is taken up for every student (row), whether they have two or not. This can lead to a big waste of disk space.
- When you search for the supervisor, which field do you look up?

Many-to-Many Relationship

We solve this by introducing a new table to link `student` and `supervisor`:



student
id
first_name
last_name
city_id
status_id

student_to_supervisor
student_id
supervisor_id

supervisor
id
first_name
last_name
room_number

This is called a join table. Note the lack of a dedicated primary key. Here, the pairing of the `student_id` and the `supervisor_id` form a unique identifier. This is called a *joint primary key* or a *composite primary key*.

Some students have no supervisors; no space in the database is used in this case.

Note how Moody (student_id=4) has two supervisors.

Student

id	first_name	last_name
1	Cara	Rogers
2	Ori	Mejia
3	Leandra	Stevens
4	Danielle	Moody
5	Josiah	Barber
6	Wing	Gordon
7	Ryder	Schneider

Student_to_Supervisor

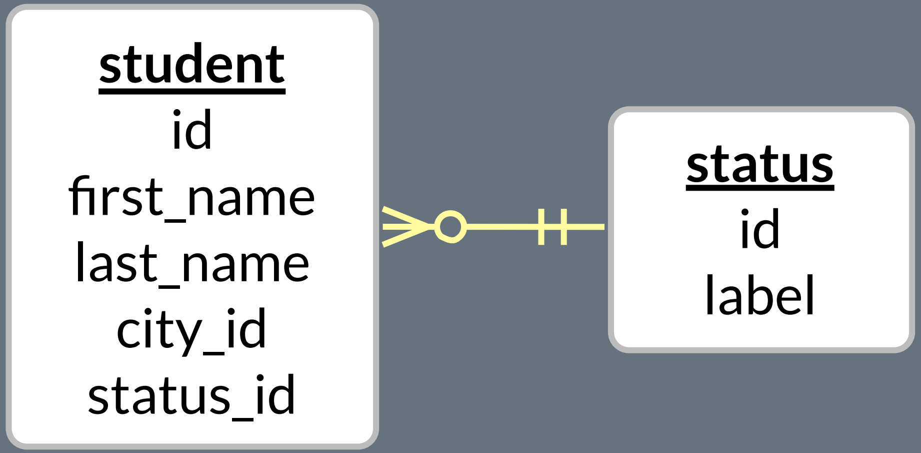
student_id	supervisor_id
1	10
4	4
4	9
6	4
7	4

Supervisor

id	first_name	last_name	room_number
10	David	Tennant	101
4	Tom	Baker	315
9	Christopher	Eccleston	205
11	Matt	Smith	210

Schema Notation

Arrows in the schema note the relationship between tables. Sometimes the information is a note to the designer and is not enforced (only indicated) in the schema design, but can be in code.

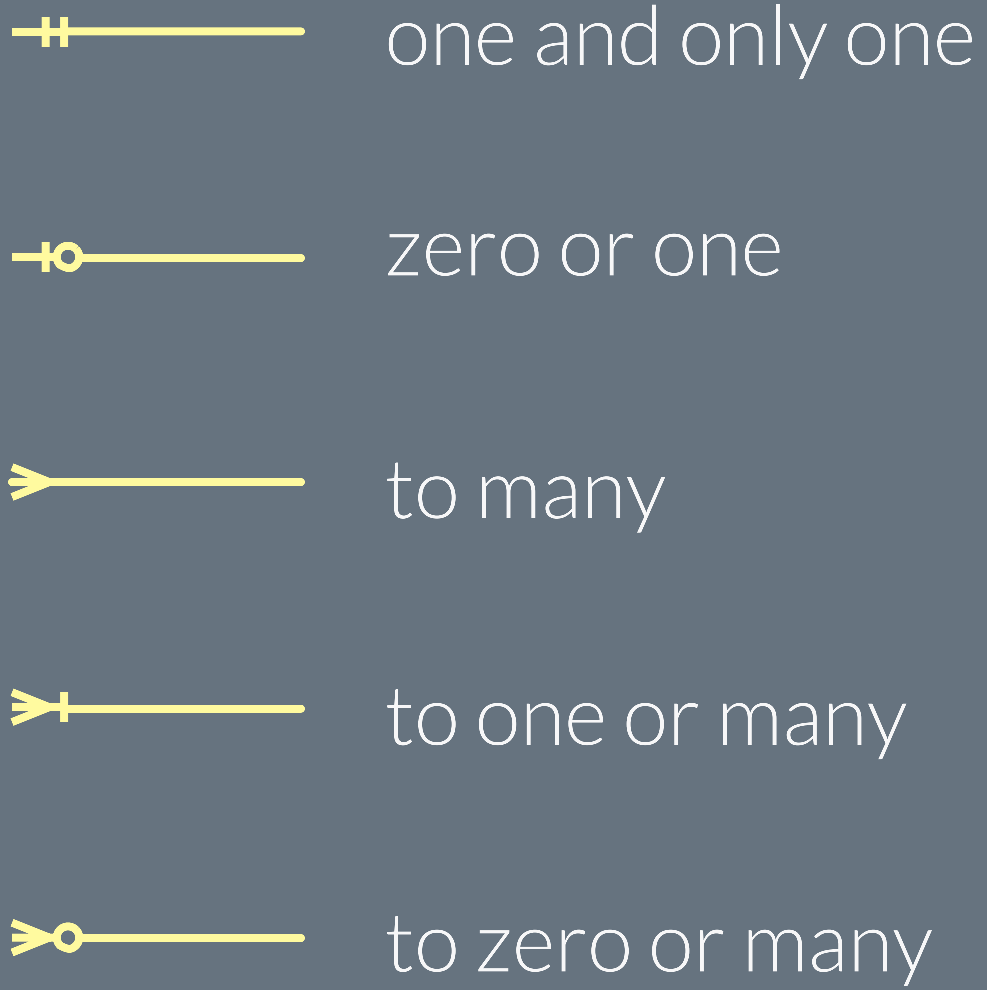


student
id
first_name
last_name
city_id
status_id

status
id
label

A student must have one and only one status (e.g. a student cannot be a freshman and a senior, and cannot be unclassified.

Many students can have a particular status (e.g. there are many sophomores). There may be no students of a single status (e.g. "senior" is a status, but there may be no seniors).



one and only one

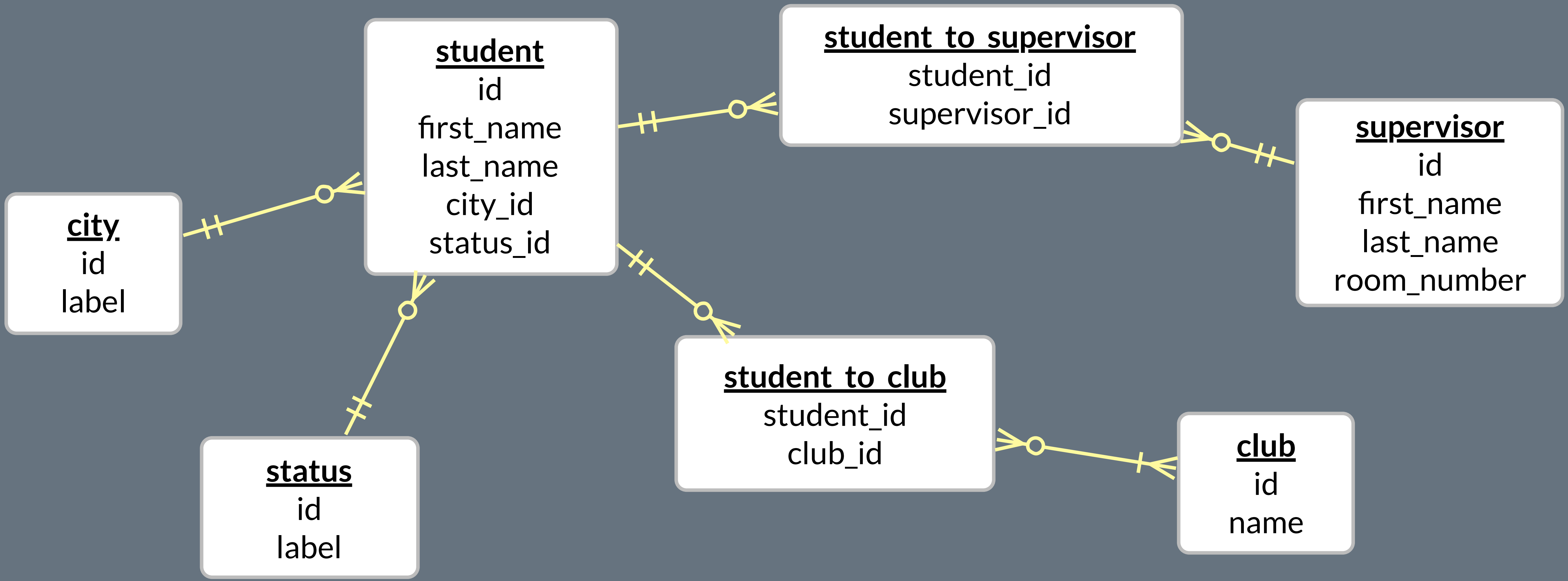
zero or one

to many

to one or many

to zero or many

The Final Schema



student
id
first_name
last_name
city_id
status_id

student_to_supervisor
student_id
supervisor_id

supervisor
id
first_name
last_name
room_number

city
id
label

status
id
label

student_to_club
student_id
club_id

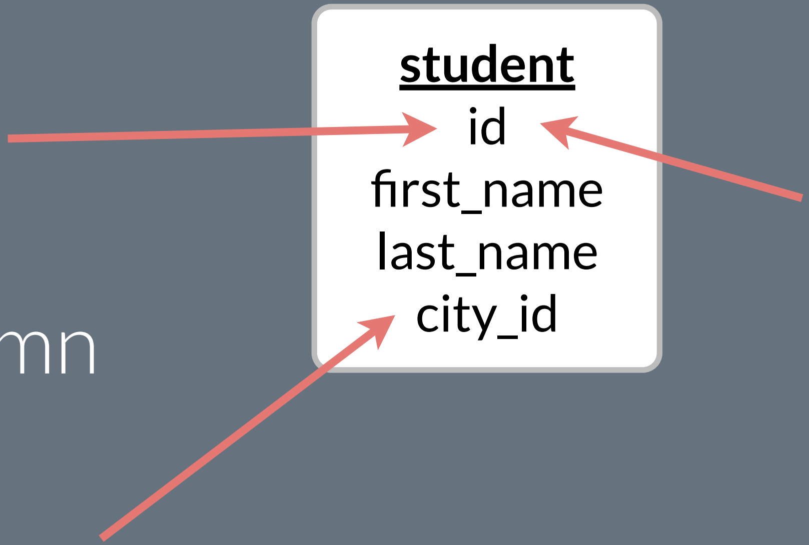
club
id
name

Constraints

Data can be validated by the program populating the data, or the database can perform some validations. The latter is always preferable.

These are called *constraints*, and there are several kinds. The database won't accept a row that fails a given constraint.

Constraints



<u>**student**</u>
id
first_name
last_name
city_id

unique constraint
all values must be unique in a given column

not NULL constraint
this column is not allowed to contain empty (NULL) values

foreign key constraint
column is a foreign key; i.e. a primary key of another column

primary key constraint
applies both a unique constraint and not NULL

Constraints can be *composite*, e.g. columns A and B together must be unique. Custom constraints can also be programmed (e.g. value must be in range [0,1]).

Primary Keys & Sequences

It's clear that primary keys should be sequential. It's tedious to have to manage them yourself: when inserting new rows, you have to:

- ask the database for the current highest value
- add one
- ask the database for the current highest value (another process might have added a row in the meantime…)
- repeat

Often you can set a primary key to *autoincrement*. If you leave the field empty (i.e. not specify it), the database will get the next value and insert it for you. You should almost always use this (the notable exception being joint primary keys in join tables).

Databases keep track of the current value in an object called a *sequence*.

Primary Keys & Sequences

Setting a field to autoincrement (or of type “serial” (integer) or “big serial” (big integer) in PostgreSQL):

- creates a new sequence for that column
- sets the initial value to 1
- gets that value when no primary key is specified for that column
- increments the sequence when a number is used

Note that if you later specify the primary key yourself, it will likely conflict with the sequence, resulting in an error later. It’s always best to let the database determine the primary key values.

Database Searches

Type of Search

Sequential Search

Database searches for a value in a column by sequentially comparing each one (default).

Indexed Search

Creating an *index* on the column dramatically speeds the search, but at the cost of disk space (that the index takes up). Thus, only index columns you will search on. (For example, *always* index foreign keys.)

Speed of Search

O(N) - this is a CS notation that means the time taken is proportional to the number of items (read: “order *n*”). Also referred to as “linear time”.

O(log(N)) - logarithmic performance, and even be as fast as **O(1)** (flat performance, i.e. search speed is the same regardless of the number of rows in the column).

Introduction to O(N) notation:
`http://www.perlmonks.org/?node_id=227909`

Exercise

Create the student database in SQLite.

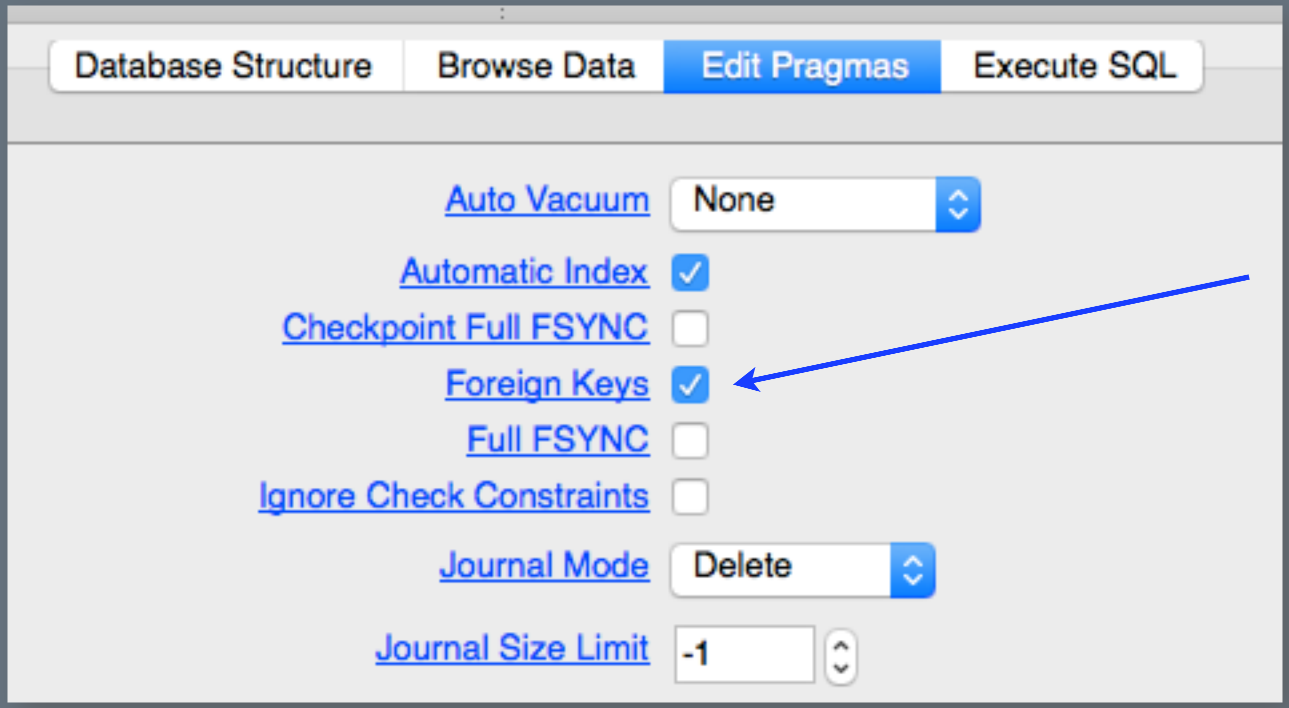
Define primary key and foreign key constraints.

Use DB SQLite Manager to create the database.

The result should be a file containing an empty database (structure, no data).

Foreign Keys In SQLite Browser

To enable foreign key support:



Auto Vacuum: None

Automatic Index ☑

Checkpoint Full FSYNC □

Foreign Keys ☑

Full FSYNC □

Ignore Check Constraints □

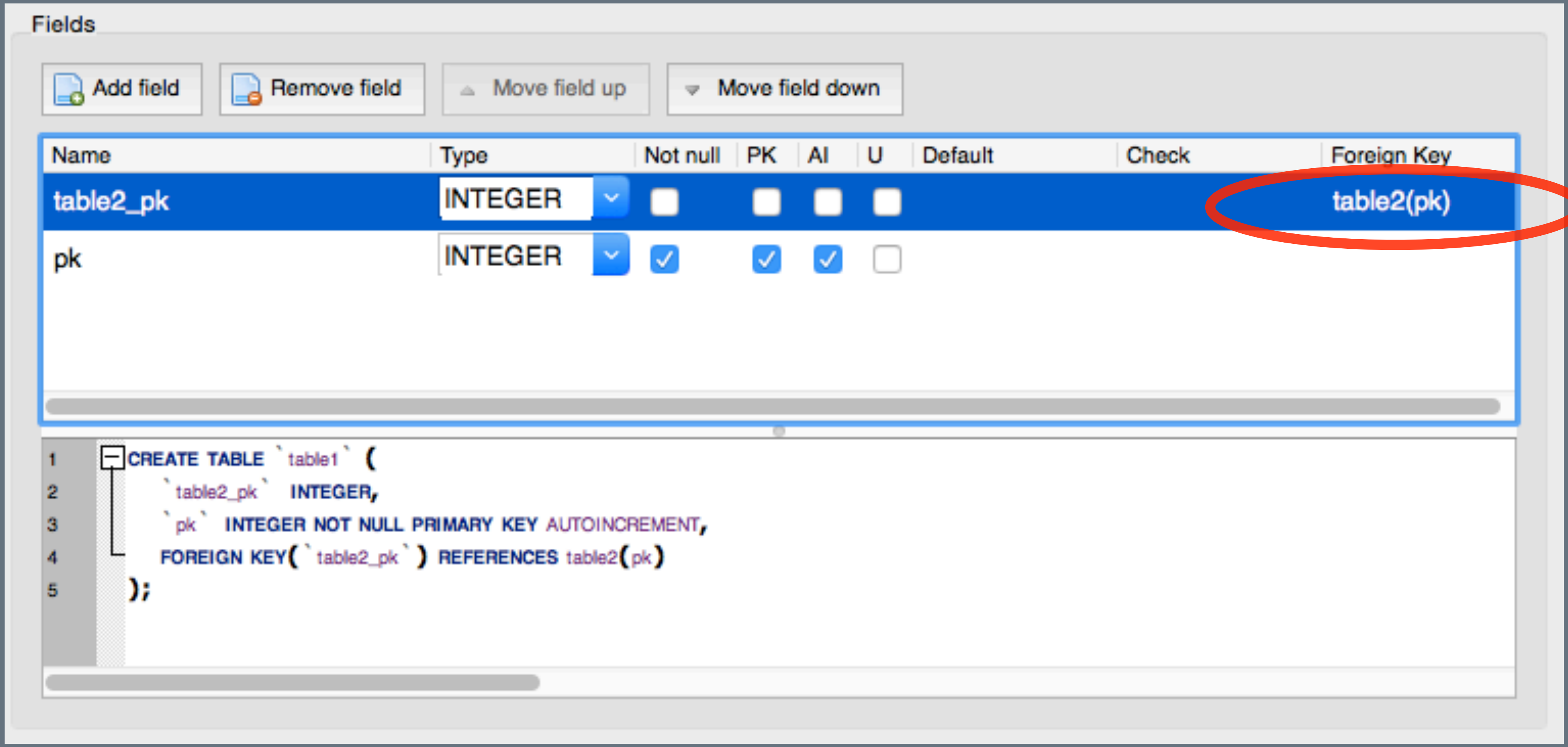
Journal Mode: Delete

Journal Size Limit: -1

check this box

Foreign Keys In SQLite Browser

To specify foreign keys:



Name	Type	Not null	PK	AI	U	Default	Check	Foreign Key
table2_pk	INTEGER	□	□	□	□			table2(pk)
pk	INTEGER	☑	☑	☑	□			

```
CREATE TABLE `table1` (
	`table2_pk`	INTEGER,
	`pk`	INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
	FOREIGN KEY(`table2_pk`) REFERENCES table2(pk)
);
```

Export Schema / Create Database

Exporting the Database

If you have an SQLite database, you can "dump" it to the raw SQL commands needed to fully recreate it:

```
% sqlite3 my_database.sqlite '.dump' > my_database_backup.sql
```

It's a good idea to make a backup of your database before adding data to it.

Creating a new database from a backup:

```
% cat my_database_backup.sql | sqlite3 new_database.sqlite
```

Note the difference between a file with raw SQL commands and a file that is the database!! They are not the same; one is used to create the other.