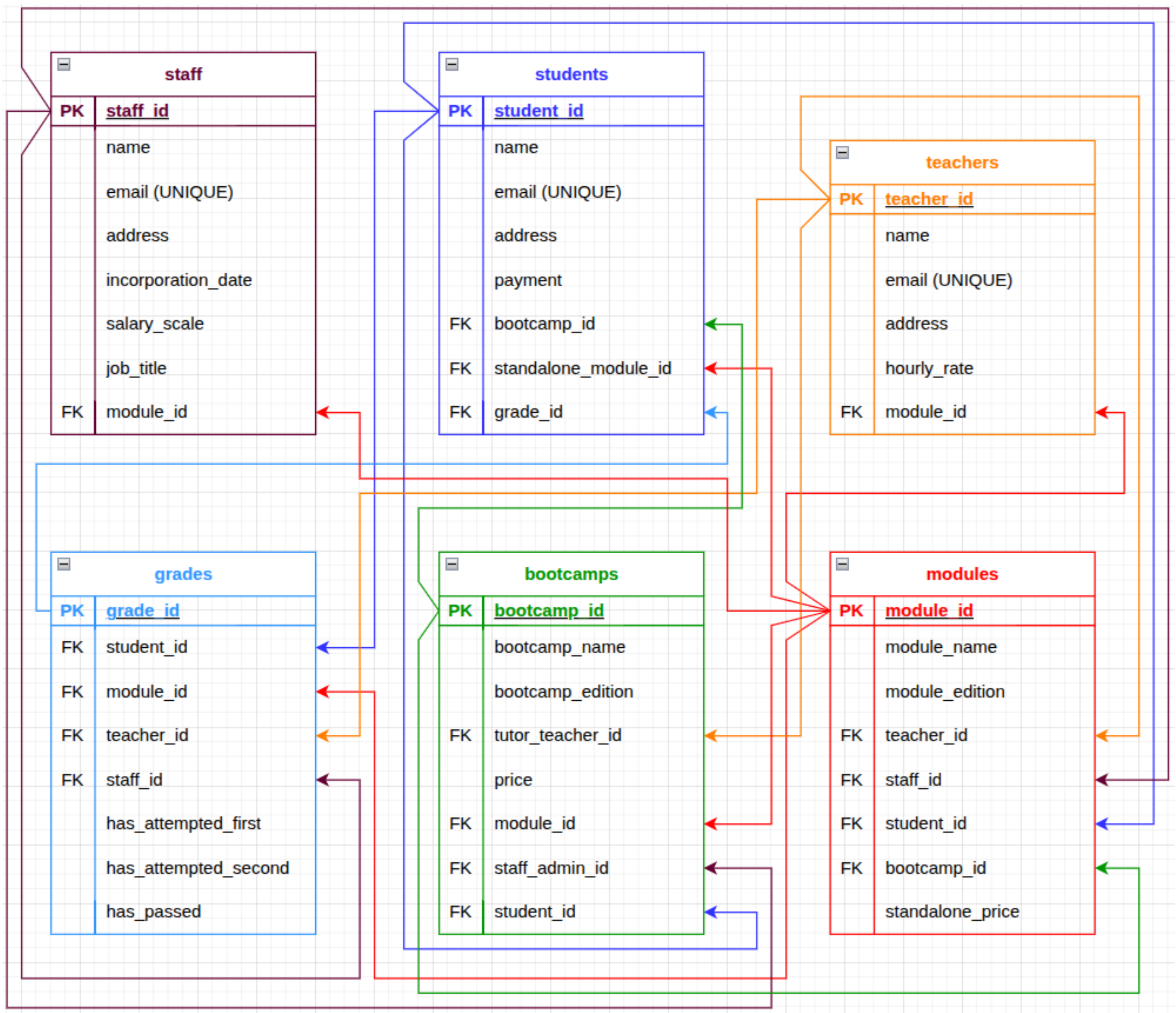
SQL Advanced Practical

by Ariane Heinz

Due 28 Sep. 2025

1) E-R Diagram for KeepCoding:



Explanation:
I have subdivided KeepCoding into six tables: three related to people and three to coursework. Each table is displayed in a different color, to facilitate legibility.

People:

- ***Staff*** refers to people who are on a salary, meaning they are working full-time for KeepCoding. It could be systems maintenance, HR, admin or even teachers who are employed as their only job. Each person is on a specific salary scale based on their job title, qualifications and contractual agreement, and I have also taken into account their incorporation year which may impact their benefits. The PK connects with *bootcamps*, *modules* and *grades*.

- ***Teachers*** refers to those people who are teaching specific modules and who are paid by the hour of instruction at a specific hourly rate. Similarly, the PK feeds *bootcamps*, *modules* and *grades* and the table is fed the module_ids that the teacher is associated with. It could be argued that this table could be combined with the *staff* table since its structure is very similar. However, I feel that for questions of access to sensitive information and data protection, staff may need privileges that should not be awarded to external teachers.
- ***Students*** obviously refers to the students taking the courses, and I placed the students at the center since without students KeepCoding ceases to exist. Again, the PK feeds into bootcamps, modules, and grades. I have planned for the possibility that students may take stand-alone modules, either in addition to a bootcamp or without being signed up to a full bootcamp. The table is fed as FK a bootcamp_id, a grade_id and a stand_alone_module_id. I also included a field for “payment” which has to do with how much the student has paid so far, so KeepCoding can keep track of what is still owed. I’m not sure how this would take place in real life, as I don’t have any experience with this.

Coursework:
- ***Modules*** is probably the most important table, as it seems to me that KeepCoding revolves around a series of modules. Thus the PK feeds into all other tables: the *teachers*, *students* and *staff* that are associated with the module, and also the *bootcamps* that use that module and the *grades*. The *modules* table is fed primary keys from all other tables except for *grades*, which only go to *students*. Modules can be taken by students as a stand_alone course or as part of a bootcamp.
- **Bootcamps** refers to a set of modules that combine to make a certification into a domain of data science. They are associated with *students*, *staff*, tutor *teachers* and *modules* and therefore receive as foreign keys the ids associated with those. The PK feeds into *students* and *modules* exclusively.
- ***Grades*** is a fairly separate and stand-alone table that receives as foreign keys the student_id, module_id, teacher_id and staff_id (mostly for cases in which the teacher is a member of staff, or in case a member of staff has to fix an error). Each grade_id (PK) only feeds into the *students* table, so the information about grades can be easily associated with each student.

Other comments:
- I could have added another table for *interested* people or *alumni*, that is people who might not yet or no longer be in the active database of KeepCoding but who may receive promotions or marketing information of some sort. However, I decided to keep it simple.
- In terms of the level of detail, I believe an actual database would have many more fields, such as separating “name” into first, middle and last name, or into first name (*nombre*) and two last names (*1r apellido, 2º apellido*), or such as including the field “phone” as a VARCHAR(20). I tried to keep it simple so the diagram does not become too difficult to read. Also, adding the extra fields would just make the exercise longer, but not teach anything meaningful, in my humble opinion.
- In terms of linking the *students* table and the *grades* table, I remember a class conversation on which way the arrow should go. I decided that student_id from the *students* table will go to *grades* but the grade_id from *grades* will go to *students*. This makes sense to me because the *grades* table is the central place where all grades are stored as a PK, and then each grade is assigned to the right student as a FK, for each module the student has coursed.
- For all people-related people, I have made the “email” field unique to ensure that there are no same emails that refer to different people, since it would be dangerous in terms of data protection to send personal information to the wrong person.

2) Database implementation in PostGreSQL:

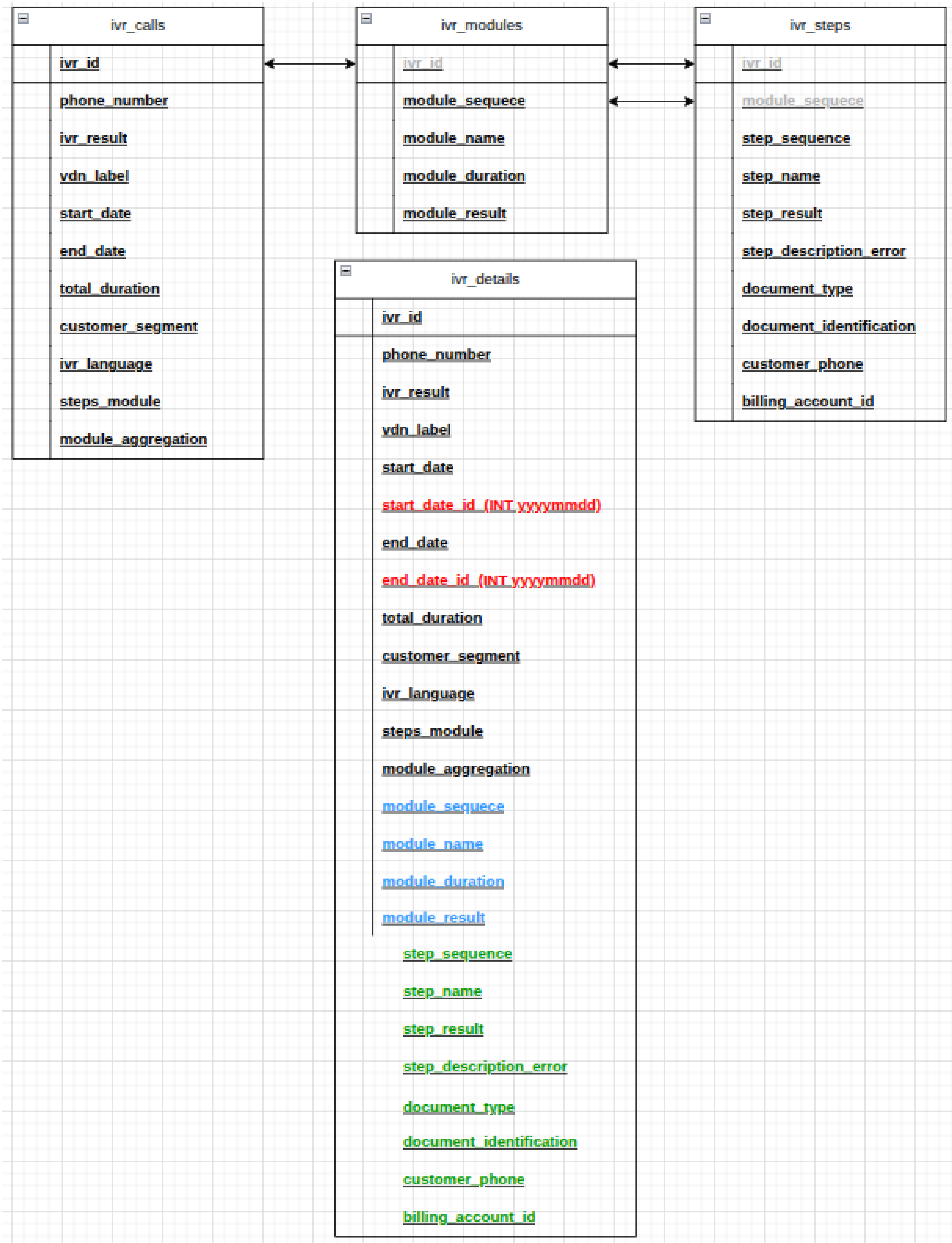
```
CREATE TABLE staff (
   staff_id SERIAL PRIMARY KEY,
   name VARCHAR (255),
   email VARCHAR (255),
   address VARCHAR (255),
   incorporation_date DATE,
   salary_scale INT,
   job_title VARCHAR (255),
   module_id INT
);
ALTER TABLE staff
ADD CONSTRAINT staff_unique_email UNIQUE (email);
ALTER TABLE staff
ALTER COLUMN email SET NOT NULL;

CREATE TABLE students (
   student_id SERIAL PRIMARY KEY,
   name VARCHAR (255),
   email VARCHAR (255),
   address VARCHAR (255),
   payment DECIMAL (8,2),
   bootcamp_id INT,
   standalone_module_id INT,
   grade_id INT
);
ALTER TABLE students
ADD CONSTRAINT students_unique_email UNIQUE (email);
ALTER TABLE students
ALTER COLUMN email SET NOT NULL;

CREATE TABLE teachers (
   teacher_id SERIAL PRIMARY KEY,
   name VARCHAR (255),
   email VARCHAR (255),
   address VARCHAR (255),
   hourly_rate DECIMAL (8,2),
   module_id INT
);
ALTER TABLE teachers
ADD CONSTRAINT teachers_unique_email UNIQUE (email);
ALTER TABLE teachers
ALTER COLUMN email SET NOT NULL;
```

```sql
CREATE TABLE grades (
   grade_id SERIAL PRIMARY KEY,
   student_id INT,
   module_id INT,
   teacher_id INT,
   staff_id INT,
   has_attempted_first BOOLEAN,
   has_attempted_second BOOLEAN,
   has_passed BOOLEAN
);

CREATE TABLE bootcamps (
   bootcamp_id SERIAL PRIMARY KEY,
   bootcamp_name VARCHAR (255),
   bootcamp_edition VARCHAR (10),
   tutor_teacher_id INT,
   price DECIMAL (8,2),
   module_id INT,
   staff_admin_id INT,
   student_id INT
);

CREATE TABLE modules (
   module_id SERIAL PRIMARY KEY,
   module_name VARCHAR (255),
   module_edition VARCHAR (10),
   teacher_id INT,
   staff_id INT,
   student_id INT,
   bootcamp_id INT,
   standalone_price DECIMAL (8,2)
);

ALTER TABLE staff
ADD CONSTRAINT fk_staff_module_id
   FOREIGN KEY (module_id) REFERENCES modules(module_id);

ALTER TABLE students
ADD CONSTRAINT fk_students_bootcamp_id
   FOREIGN KEY (bootcamp_id) REFERENCES bootcamps(bootcamp_id),
ADD CONSTRAINT fk_students_standalone_module_id
   FOREIGN KEY (standalone_module_id) REFERENCES modules(module_id),
ADD CONSTRAINT fk_students_grade_id
   FOREIGN KEY (grade_id) REFERENCES grades(grade_id);
```

```sql
ALTER TABLE teachers
ADD CONSTRAINT fk_teachers_module_id
   FOREIGN KEY (module_id) REFERENCES modules(module_id);

ALTER TABLE grades
ADD CONSTRAINT fk_grades_student_id
   FOREIGN KEY (student_id) REFERENCES students(student_id),
ADD CONSTRAINT fk_grades_module_id
   FOREIGN KEY (module_id) REFERENCES modules(module_id),
ADD CONSTRAINT fk_grades_teacher_id
   FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id),
ADD CONSTRAINT fk_grades_staff_id
   FOREIGN KEY (staff_id) REFERENCES staff(staff_id);

ALTER TABLE bootcamps
ADD CONSTRAINT fk_bootcamps_tutor_teacher_id
   FOREIGN KEY (tutor_teacher_id) REFERENCES teachers(teacher_id),
ADD CONSTRAINT fk_bootcamps_module_id
   FOREIGN KEY (module_id) REFERENCES modules(module_id),
ADD CONSTRAINT fk_bootcamps_staff_admin_id
   FOREIGN KEY (staff_admin_id) REFERENCES staff(staff_id),
ADD CONSTRAINT fk_bootcamps_student_id
   FOREIGN KEY (student_id) REFERENCES students(student_id);

ALTER TABLE modules
ADD CONSTRAINT fk_modules_teacher_id
   FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id),
ADD CONSTRAINT fk_modules_staff_id
   FOREIGN KEY (staff_id) REFERENCES staff(staff_id),
ADD CONSTRAINT fk_modules_student_id
   FOREIGN KEY (student_id) REFERENCES students(student_id),
ADD CONSTRAINT fk_modules_bootcamp_id
   FOREIGN KEY (bootcamp_id) REFERENCES bootcamps(bootcamp_id);
```

3) Creating table of ivr_detail:

First I created an E-R diagram of the tables to understand the question a little bit better (see next page).



Then I created some queries to understand the meaning of the table a little bit better, and to see whether I would need a FULL OUTER JOIN, an INNER JOIN, or a LEFT JOIN, as well as to estimate the number of rows of my created table.

I understand that table *ivr_calls* contains data about the calls themselves, and that table *ivr_modules* information about the modules that each call goes through. Thus I expect each call to have several rows in the *ivr_modules* table. In my first query I am finding out on average how many modules each call goes through. The answer is 6.2 modules per call on average, calculated from the 21674 distinct calls. Next I also check how many modules do not have a call associated, and I find that the answer is none.

```
-- How many modules per call, on average? And how many distinct calls?
SELECT
   COUNT(cal.ivr_id) AS total_calls,
   COUNT(DISTINCT cal.ivr_id) AS distinct_calls,
   AVG(COALESCE(mod.mod_count,0)) AS avg_modules_per_call
FROM keepcoding.ivr_calls as cal
LEFT JOIN (
   SELECT ivr_id, COUNT(*) AS mod_count
   FROM keepcoding.ivr_modules
   GROUP BY ivr_id
) as mod
ON cal.ivr_id = mod.ivr_id;

-- How many modules do not have a call?
-- For this I do a LEFT JOIN FROM "ivr_modules" ON "ivr_calls".
SELECT COUNT(*) AS modules_without_call
FROM keepcoding.ivr_modules as mod
LEFT JOIN keepcoding.ivr_calls as cal
ON mod.ivr_id = cal.ivr_id
WHERE cal.ivr_id IS NULL;
```

In my next query, similarly, I want to see how many steps, on average, take place within each module. The answer is that there are an average of almost 42800 steps per module, and a total of 133599 modules, of which only 8 are distinct.

With this information, I decided to use a FULL OUTER JOIN to create the table, knowing very well that it would make a gigantic table, since each call had an average of over 6 modules, and each module an average of almost 42800 steps.

Thus I was expecting a table that could have 21674 x 6.2 x 42800 <= 5.751.412.640 rows! But in the end it was “only” 1.909.080 rows.

The code is as follows:

```
CREATE TABLE keepcoding.ivr_detail AS

WITH date_ids AS (
   SELECT
       cal.start_date,
       DATE(cal.start_date) AS start_date_id,  --if leaving as "date"
       --FORMAT_DATE('%Y%m%d', DATE(cal.start_date)) AS start_date_id, --if
changing to "string"
       cal.end_date,
       DATE(cal.end_date) AS end_date_id,  --if leaving as "date"
       --FORMAT_DATE('%Y%m%d', DATE(cal.end_date)) AS end_date_id, --if changing to
"string"
       cal.ivr_id
   FROM keepcoding.ivr_calls as cal
)
SELECT
```

```sql
   cal.ivr_id,
   cal.phone_number,
   cal.ivr_result,
   cal.vdn_label,
   cal.start_date,
   start_date_id,
   cal.end_date,
   end_date_id,
   cal.total_duration,
   cal.customer_segment,
   cal.ivr_language,
   cal.steps_module,
   cal.module_aggregation,
   mod.module_sequece,
   mod.module_name,
   mod.module_duration,
   mod.module_result,
   ste.step_sequence,
   ste.step_name,
   ste.step_result,
   ste.step_description_error,
   ste.document_type,
   ste.document_identification,
   ste.customer_phone,
   ste.billing_account_id
FROM keepcoding.ivr_calls as cal
FULL OUTER JOIN date_ids
   ON cal.ivr_id = date_ids.ivr_id
FULL OUTER JOIN keepcoding.ivr_modules as mod
   ON cal.ivr_id = mod.ivr_id
FULL OUTER JOIN keepcoding.ivr_steps as ste
   ON cal.ivr_id = ste.ivr_id;
```