

An informal introduction to relational databases

GRA 4142 Data Management and Python Programming, Fall 2021

Jan Kudlicka (jan.kudlicka@bi.no)

This week we will start learning SQL—Structured Query Language, a special-purpose language designed for querying and managing data in relational databases. But what exactly are relational databases? And why should we bother to use them instead of Pandas? Below we will introduce relational databases informally, and briefly compare them to Pandas data frames. In a few weeks we will return to this topic, and cover the relational model more thoroughly.

A relational database

A relational database is a collection of named tables. Each table, or **relation** how it is called in the relational model, consists of columns (**attributes, fields**) and rows (**tuples, records**).

Just as in Pandas, each attribute (column) has a unique name and a data type (see the list of the most common data types in the appendix). Both have to be specified when creating the table. For each attribute we can also specify additional constraints, for example if missing values (denoted by a reserved word **NULL**) are not allowed, or if the values must be unique.

Note that, unlike in Pandas, tuples (rows) are not ordered.

Primary key

Each table has a **primary key** (PK)—an attribute (or a set of several attributes) that uniquely identifies each tuple. For example, consider a table of employees in a company database with attributes SSN (social security number), first name, last name, email and salary. Assuming that SSN cannot contain NULLs, we can use this attribute as the primary key: it uniquely identifies all employees since different employees must have different social security number.

In practice we often choose to add a special ID attribute to be used as the PK. Many database systems support *auto increment* attributes that will automatically set the value to a unique numeric ID when a new row is inserted into a table (usually starting at 1 and increased by 1 for each new row).

Using several relations

Imagine that BI has asked us to create a simple database to store exam grades. Our first design is a single table with the following attributes:

```
Student_ID, Student_FirstName, Student_LastName, CourseCode, CourseTitle,
Grade, Date
```

Can you see some problems related to this design? There are quite a few of them:

- **Insertion anomaly:** Assuming that grade and date are mandatory, we cannot store information about students before their first exam. The same applies to courses.
- **Deletion anomaly:** If we (for some reason) remove all exam grades for a given student, the information about that student is lost as well. The same applies to courses.

- **Update anomaly:** When some information changes (e.g. the surname of a student who got married), we need to update it on all places where it is stored; failing to do so will introduce data inconsistency.
- And finally, **redundancy** (related to the update anomaly): Same information is stored multiple times, which is a waste of storage and a risk of inconsistency.

Each student takes many courses and each course has many students (in other words, there is a many-to-many relationship between students and courses), so the information like the first and last name, and the course title will be repeated many times. It is much better to move the information about students to a separate table (using the student ID as the PK):

`Student(ID, First_Name, Last_Name)`

Similarly we can move the information about courses to a separate table too (using the code as the PK):

`Course(Code, Title)`

The PKs we have chosen allow us to identify each student by their ID and each course by its code. The grade table can be now simplified to:

`Grade(Student_ID, Course_Code, Grade, Date)`

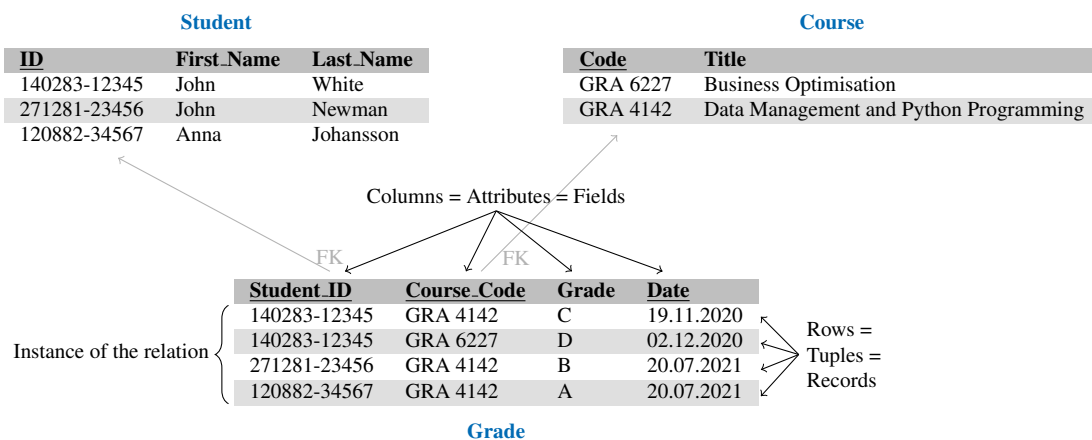
Exercise: What can we use as the PK in the Grade table?

If we look at a particular tuple (row) in the Grade table, say (120882-34567, GRA 4142, A, 20.07.2021), and want to know the student's name, we can just look up the student ID (120882-34567) in the Student table. Since the student ID is the PK in this table, we know that there cannot be several students with this ID. But we would also like to make sure that the information about the student does exist in the table!

Foreign key

Foreign key (FK) is an attribute (or a set of attributes) that refers to the primary key of another relation (table). The foreign key value, if it is given, must match the primary key value of an existing tuple (row) in the other relation (table).

A foreign key creates a link between rows in two tables. Note that the we can also reference rows in the same table, e.g. in the table of employees we might want to store the supervisor for each employee (and the supervisor is also an employee).



In our example, we want the `Student_ID` attribute in the `Grade` table to be a foreign key referencing the PK in the `Student` table, and the `Course.Code` attribute to be (another) FK referencing the PK in the `Course` table (see the figure above). This will ensure that we cannot insert a row into the `Grade` table that uses a non-existing student ID and/or a non-existing course code, and also makes sure that we cannot remove students or courses if there exist referencing tuples in the `Grade` table (this is called *referential integrity*).

Pandas data frames vs. relational databases

Data structures in Python, including Pandas data frames, are kept in volatile memory, i.e. memory that requires power to maintain the data stored in it. Reading and writing data from and to volatile memory is fast, but the data is lost when a program finishes or crashes due to a bug, or when the computer loses power or crashes due to another hardware problem. In addition, we often need to work with data sets that do not fit into the volatile memory (and since the volatile memory is expensive, buying more is not an option). Pandas have limited possibilities when it comes to the integrity and consistency of the data, and finally, the data frames cannot be (in an easy way) accessed by multiple users or applications at the same time.

Relational databases are stored in non-volatile memory (e.g. hard disk drives, solid state drives) and we can therefore work with (really) big databases. Reading and writing data from and to non-volatile memory is however slow, so database systems must be very efficient in how the data is stored and retrieved. Database systems also ensure the integrity, consistency and validity of data, even in situations where many users are accessing and modifying the database concurrently.

Non-relational databases

There are also non-relational databases, often called NoSQL databases (NoSQL meaning non-SQL, non-relational, not only SQL). Examples of these include:

- Key-value stores
- Document-oriented databases
- Graph databases
- etc.

NoSQL databases are popular in e.g. big data and real-time web applications.

Appendix. The most common data types in relational databases

- BOOLEAN
- INT
- FLOAT
- DOUBLE
- DECIMAL (p, s) , exact numeric value with p significant digits and s digits after the decimal point.
- CHAR (l) , fixed-length string of length l . (Max. value for l is usually 255.)
- VARCHAR (l) , variable-length string of max. length l . (Max. value for l is usually 255.)
- ENUM $(s_1, s_2, \dots, s_n)$, value from a set of allowed strings.
- TEXT
- DATE
- DATETIME

Note: Support for data types varies slightly between different database systems, so check the documentation for the system you use.