Relational databases
Description of methodology
Based on a course offered by DataCamp
Introduction to Relational Databases[1]
and other sources

Stepan Oskin

August 2, 2019

Abstract

A relational database models real life entities, such as universities and university professors, by storing them in tables. The idea of a database is to push data into a certain structure —a pre-defined model —where you can enforce data types, relationships, and other rules. Each table must contain data from a single entity type. This reduces redundancy by storing entities only once. A database can then be used to model relationships between entities and to preserve data quality through such concepts as constraints, keys, and referential integrity. SQL, or Structured Query Language, can be used for querying, as well as building and maintaining databases.

1 Relational Database

A relational database models real life entities, such as universities, organizations, and university professors, by storing them in tables. Each table must contain data from a single entity type (*e.g.*, universities, organizations, university professors, *etc.*) This reduces redundancy by storing entities only once —for example, there only needs to be one row of data containing details of a certain university. Lastly, a database can be used to model relationships between entities. For instance, a professor could be involved with multiple organizations within a university, while an organization may include many professors.

An example of data organized with redundancy in attributes and without redundancy is presented in the two entity-relationship diagrams on figure 1. The diagram on the left (fig.1a) models only one entity type —`university_professors`. However, this table actually holds multiple entity types, and thus causes redundancy in entries, as the same professor and university can be present in multiple rows, for example, in cases where the professor is involved with multiple organizations within the same university. The updated entity-relationship model (fig.1b) would be better suited in this case. It represents three entity types —"professors", "universities", and "organizations" —in their own tables, with respective attributes. This reduces redundancy, as professors, unlike in fig. 1a, need to be stored only once. For each professor, the respective university is denoted through the `university_shortname` attribute.

Characteristics of a relational database:

- real-life *entities* become *tables*
- reduced redundancy
- data integrity by *relationships*

2 Integrity constraints

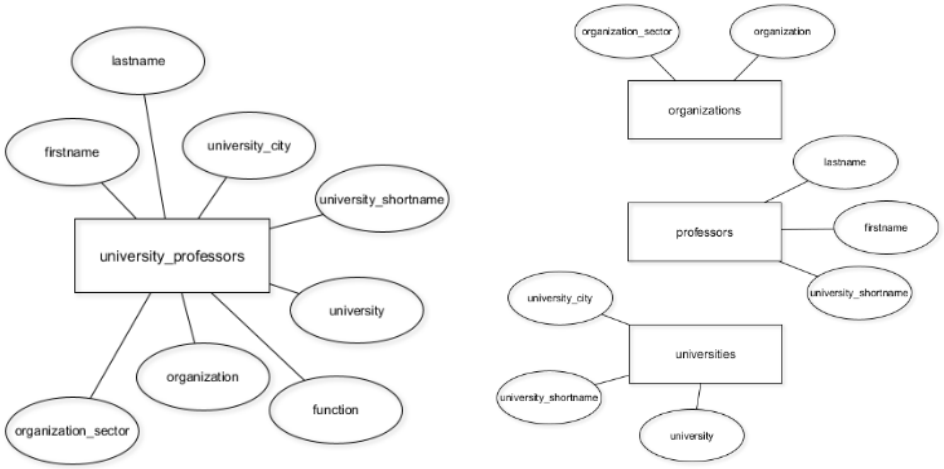
The idea of a database is to push data into a certain structure —a predefined model —where you can enforce data types, relationships, and other rules. Generally, these rules are called **integrity constraints**, although different names exist.

Integrity constraints can roughly be divided into 3 types:

1. **Attribute constraints**, *e.g.*, data types on columns
2. **Key constraints**, *e.g.,* primary keys
3. **Referential integrity constraints**, *e.g.,* enforced through foreign keys

Benefits of using constraints include:

- Constraints press data into a certain structure



(a) with redundancy (b) without redundancy

Figure 1: These two entity-relationship diagrams show examples of organizing data with redundancy in attributes (fig. 1a), and without redundancy (fig. 1b). Squares denote the so-called entity types, and circles connected to these denote attributes (or columns).

- Constraints help with consistency, and thus data quality (*e.g.,* by ensuring that the certain format is followed during manual or automated data entry)
- Data quality is a business advantage / data science prerequisite
- Enforcing constraints is difficult, but database management systems (*e.g.,* PostgreSQL) help

2.1 Attribute constraints

In its simplest form, attribute constraints are data types that can be specified for each column in a table. There are basic data types for numbers, such as `bigint`, or strings of characters, such as `character varying`. There are also more high-level data types, such as `cidr`, which can be used for IP addresses. Implementing such a data type on a column would disallow anything that doesn't fit the structure of an IP .

Data types also restrict possible SQL operations on the stored data. For example, it is impossible to calculate a product from an `integer` and a `text` column. In case if a column with data type `text` contains numbers

that need to be used in calculations, on-the-fly type conversions, or **type casts** can be used to allow the required operation using the `CAST` function in SQL . Examples of some data types from *PostgreSQL documentation*[2] are presented on figure 2.

Name	Aliases	Description
`bigint`	`int8`	signed eight-byte integer
`bigserial`	`serial8`	autoincrementing eight-byte integer
`bit [ (n) ]`		fixed-length bit string
`bit varying [ (n) ]`	`varbit [ (n) ]`	variable-length bit string
`boolean`	`bool`	logical Boolean (true/false)
`box`		rectangular box on a plane
`bytea`		binary data ("byte array")
`character [ (n) ]`	`char [ (n) ]`	fixed-length character string
`character varying [ (n) ]`	`varchar [ (n) ]`	variable-length character string
`cidr`		IPv4 or IPv6 network address

Figure 2: Beginning of the list of all data types from PostgreSQL documentation[2].

2.1.1 Data types

Working with data types can be straightforward in a database management system such as PostgreSQL . As said before, data types are attribute constraints and are therefore implemented for a single column of a table. They define the so-called *domain of values* in a column, that means, what form these values can take —and what not. Therefore, they also define what operations are possible with the values in the column. Through this, consistent storage is enforced, so a street number will always be an actual number, and a postal code will always have exactly 6 characters, according to the defined conventions. This greatly helps with data quality.

To sum up, data types in a database management system are:

- Enforced on columns (*i.e.,* attributes)
- Define the so-called *domain* of a column
- Define which operations are possible
- Enforce consistent storage of values

Most common data types in PostgreSQL include:

- `text`: character string of any length
- `varchar [ (n) ]`: a maximum of `n` characters
- `char [ (n) ]`: a fixed-length string of `n` characters
- `boolean`: can only take three states, *e.g.,* `TRUE`, `FALSE`, and `NULL` (unknown)
- `date`, `time`, and `timestamp`: various formats for date and time calcualtions
- `numeric`: arbitrary precision numbers, *e.g.,* 3.1457
- `integer`: whole numbers in the range of -2'147'483'648 and +2'147'483'647
- `bigint`: whole numbers in a larger range

These data types are specific to PostgreSQL, but appear in many other database management systems as well, and they mostly conform to the SQL standard.

2.1.2 The `not-null` constraint

Another type of attribute constraints are the **not-null** constraints. The not-null constraint disallows any `NULL` values on a given column. This must hold true for the existing state of the database, as well as for the future state. Therefore, we can only specify a not-null constraint on a column that doesn't hold any `NULL` values yet. And, it won't be possible to insert any null values in the future.

There is no clear definition of what do `NULL` values actually mean. `NULL` can mean a couple of things, for example, that the value is unknown, or does not exist at all. It can also be possible that the value does not apply to the column. One important takeaway is that two `NULL` values must not have the same meaning. This also means that comparing `NULL` with `NULL` always results in a `FALSE` value.

To sum up, the not-null constraints:

- Disallow `NULL` values in a certain column.
- Must hold true for the current state

- Must hold true for any future state
- `NULL` can mean different things: does not exist, does not apply, missing, *etc.*
- Therefore, `NULL != NULL`

2.1.3 The `unique` constraint

The unique constraint on a column makes sure that there are no duplicates in a column. So, any given value in a column can only exist once. This makes sense in cases where storing values more than once leads to unnecessary redundancy. However, it does not make sense in cases where it is expected that values might repeat. Just as with the not-null constraint, we can add a unique constraint if the column does not have any duplicates before we apply it. Making sure that a column contains only unique values is a prerequisite for turning them into *primary keys*

To sum up, unique constraint:

- Disallows duplicate values in a column
- Must hold true for the current state
- Must hold true for any future state
- Is a prerequisite for turning the column into a primary key

2.2 Key constraints

Key constraints are a very important concept in database systems. Attribute constraints discussed in subsection 2.1, such as data types, not-null, and unique constraints, do not change the structure of the database model. Key constraints, such as primary keys, do make a change in the model and, thus, are reflected in the entity-relationship diagram. Attributes corresponding to keys are denoted by underlying the attribute names.

Typically, a database table has an attribute, or a combination of multiple attributes, whose values are unique across the whole table. Such attributes identify a record in this table uniquely. Normally, a table, as a whole, only contains unique records, meaning that the combination of all attributes is a key in itself. However, it's not called a key, but a **superkey**, if attributes from that combination can be removed, and the attributes still uniquely identify records. If all possible attributes have been removed, but the records

are still uniquely identifiable by the remaining attributes, we speak of a minimal superkey. This is the actual key. So a key is always minimal. Minimal superkeys are also called candidate keys. In the end, there can only be one key for a table, which has to be chosen from candidates.

To sum up the information about keys, in a database:

- A key is an attribute/multiple attributes that identify a record uniquely
- As long as attributes can be removed: **superkey**
- If no more attributes can be removed: minimal superkey, or **key**
- Minimal superkeys are also called candidate keys
- Only one candidate key can be the *chosen* key

2.2.1 Primary keys

Primary keys are one of the most important concepts in database design. Almost every database table should have a primary key, chosen from a set of candidate keys. The main purpose of a primary key is to uniquely identify records in a table, and ideally, primary keys consist from as few columns as possible. For instance, this makes it easier to reference these records from other tables. Primary keys need to be defined on columns that don't accept duplicate or null values. Lastly, primary key constraints are time invariant, meaning that they must hold for the current data in the table —but also for any future data a table might hold. It is therefore wise to choose columns where values will always be unique and not null.

A table can have one and only one primary key. It is a good practice to add a primary key to every table. When we add a primary key to a table, PostgreSQL creates a unique B-tree index on the column or a group of columns used to define the primary key.[3]

To sum up the information about primary keys, in a database:

- One primary key per database table, chosen from candidate keys
- Uniquely identifies records, *e.g.,* for referencing in other tables
- Unique and not-null constraints both apply
- Primary keys are time-invariant and therefore must be chosen wisely
- Ideally, primary keys consist from as few columns as possible
- It is a good practice to add a primary key to every table.

2.2.2 Surrogate keys

Surrogate keys are sort of an artificial primary key. In other words, they are not based on a native column in the data, but on a column that just exists for the sake of having a primary key. There are several reasons for creating an artificial surrogate key. As mentioned in subsection 2.2.1, a primary key is ideally constructed from as few columns as possible. Secondly, a primary key of the record should never change over time. If we define an artificial primary key, ideally consisting of a unique number or string, we can be sure that this number stays the same for each record. Other attributes might change, but the primary key always has the same value for a given record.

There is a special data type in PostgreSQL that allows the addition of auto-incrementing numbers to an existing table: the `serial` type. It is specified just like any other data type. Once we add a column with a `serial` type, all the records in the table would be numbered. Whenever a new record is added to the table, it will automatically be given a number that does not exist yet. There are similar data types in other database management systems, like MySQL . Also, if we try to specify an ID that already exists, the primary key constraint will prevent us from doing so.

Another strategy for creating a surrogate key is to combine two existing columns into a new one. The `CONCAT` function glues together the values of two or more existing columns. After creating a new column and updating it with concatenated values, we can turn the new column into a surrogate primary key.

To sum up the information about surrogate keys, in a database:

- Primary keys should be built from as few columns as possible
- Primary keys should never change over time
- When this conditions are not met by native columns in a table, surrogate keys can be used as artificial primary keys
- For example, using the `serial` data type in PostgreSQL would assign a unique, non-null, auto-incrementing id to each record in a table
- Alternatively, surrogate keys can be created by concatenating several existing columns using the `CONCAT` function.

2.3 Referential integrity constraints

Now, we will model the relationships between professors and universities. As we know, in our database each professor works for a university. In the ER

diagram (presented on figure 3), this is drawn with a rhombus. The small numbers specify the cardinality of the relationship: a professor works for at most one university, while a university can have any number of professors working for it —even zero. Such relationships are implemented with **foreign keys**.

Foreign keys are designated columns that point to a primary key of another table. There are some restrictions for foreign keys. First, the domain and the data type of a foreign key must be the same as one of the primary key to which it is linked to. Secondly, only foreign key values that exist as values in the primary key of the referenced table are allowed. This is the actual foreign key constraint, also called **referential integrity**. Lastly, a foreign key is not necessarily an actual key, because duplicates and `NULL` values are allowed.

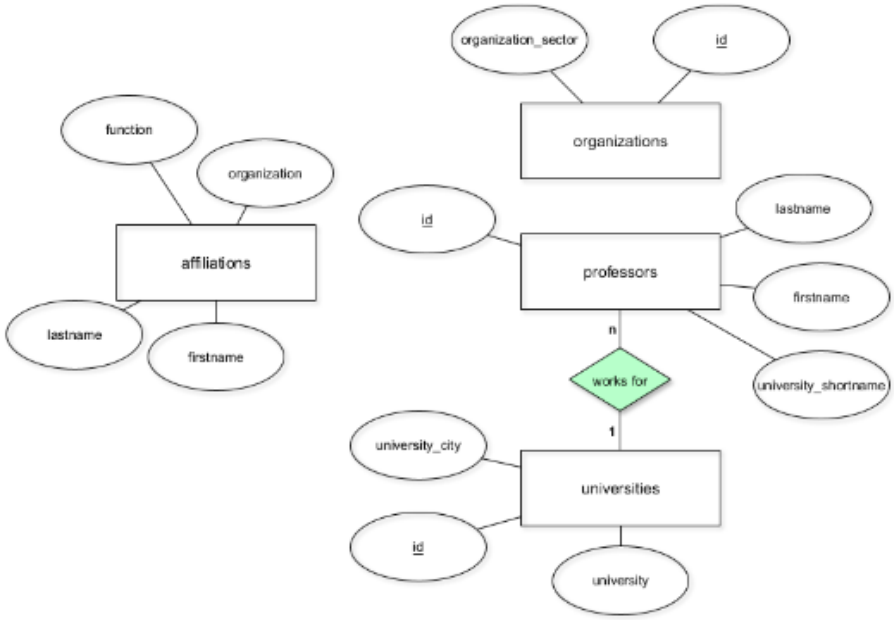


Figure 3: In the entity-relationship (ER) diagram, relationships between entities are drawn with a rhombus. The small numbers specify the cardinality of the relationship: in this database, a professor works for at most one university, while a university can have any number of professors working for it —even zero.

To sum up the information about foreign keys, in a database:

- A foreign key (FK) points to a primary key (PK) of another table
- Domain of FK must be equal to domain of PK
- Each value of FK must exist in PK of the other table (FK constraint, or **referential integrity**)
- FKs are not actual keys —`NULL` and duplicate values are allowed

3 `information_schema` database

`information_schema` database is available by default in PostgreSQL and presents a *meta database* that holds information about a relational database in PostgreSQL . `information_schema` is not PostgreSQL specific, and is also available in other database management systems, such as MySQL or SQL Server. The `information_schema` database holds various information in different tables, for example, in a `tables` or `columns` tables.

3.1 Querying `information_schema` using `SELECT * FROM` syntax

SQL, or **Structured Query Language**, can be used for querying, as well as building and maintaining databases. `information_schema` has multiple tables you can query with the known `SELECT * FROM` syntax:

- `tables`: information about all tables in your current database
- `columns`: information about all columns in all of the tables in your current database
- ...

Example: get information on all table names in the current database, while limiting your query to the 'public' `table_schema`.

```
   -- Query the right table in information_schema
SELECT table_name
FROM information_schema.tables
-- Specify the correct table_schema value
WHERE table_schema = 'public';
```

Example: get information on all column names in a particular table, while limiting your query to the 'public' `table_schema`.

```
-- Query the right table in information_schema to get columns
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = 'nhl_draft' AND table_schema = 'public';
```

References

[1] T. Grossenbacher, " Introduction to Relational Databases in SQL Course ," 2019.

[2] The PostgreSQL Global Development Group , " PostgreSQL 11 - Documentation - Data Types ," 2019.

[3] The PostgreSQL Global Development Group, "PostgreSQL Primary Key," 2019.