Introduction to SQL and Databases

Databases, DBMS, and SQL

Spreadsheets and `.csv` files are great solutions for some data, but if you need to store a large amount of data or analyse it in complex ways, they aren't the best option. As spreadsheets get larger, they get much slower, and it's hard to form key connections between different datasets without causing problems and creating lots of extra work.

When you have a large volume of complex data, the best solution is a *database*. They might have a reputation for being a little dry, but databases are fascinating, and in this article, we're going to introduce you to some of the key concepts involved in the surrounding technologies.

The most basic definition of a ***database*** is somewhat all-encompassing: a collection of tables, combined with an account of how those tables relate to one another. That's a broad definition, and does cover spreadsheet software, but that's not how the term "database" is normally used. Generally, when we use the word "database", we're actually referring to software (***DBMS*** - a Data**b**ase **M**anagement **S**ystem) running on a dedicated machine (a server) or running on your own machine (a local instance). You can then use tools to connect to this DBMS to extract data.

One of the most common types of database is a Relational database (giving us an ***RDBMS***). This is a database in which data is stored in a series of tables with links connecting data from one table with data in another, like an Excel spreadsheet with multiple worksheets. Edgar F. Codd in the 1970s wrote a paper on the theory of these databases, tying them to something called 'relational algebra' and ultimately, set theory. You'll look at this set theoretical underpinning of RDBMS in a video later in this subunit!

There are actually quite a few different DBMS and RDBMS: you might have heard, for example, of *MySQL*, *Microsoft SQL Server* and *PostgreSQL*. There's no one 'best' DBMS, and each has its own merits and demerits. Check out this link, if you're interested, for a comparison of some of the most widely used. All Database Management Systems have some form of ***transaction management***, however. This is a neat program that ensures that the database is left in a consistent state when users perform ***transactions*** (essentially just updates, retrievals and deletions) on it.

SQL**,** by contrast, is a database-querying language; a language we can use both to manipulate the data within pre-existing data structures, and to create data structures themselves. It has its own syntax, much like R and Python, but is especially forgiving in its rules; making it one of the most sweetly satisfying and immediately applicable languages in the budding data scientist's arsenal. The admissible syntax of SQL can also depend on the DBMS being used, and admissible SQL in MySQL is slightly different to that, for example, in PostgreSQL.

Different (R)DBMS have different Integrated Development Environments (***IDE***s) with which they can be used - where an IDE is essentially a coding environment in which we can execute SQL queries on our database and easily see the resultant tables. The alternative to using an IDE is executing SQL queries in the command line, and seeing the output there (which, take it from me, is significantly less pretty)! Such IDEs include *MySQLWorkbench* (for use with, no surprises, the MySQL DBMS), *pgAdmin* (for use with PostgreSQL) and *phpMyAdmin* (for use with MySQL).

With SQL and an RDBMS at our disposal, ***in-database analytics*** is possible; that is, we can do some data analysis within the database. A key advantage of this is that the necessity of moving data into an analytic tool is eliminated, and we can obtain almost real-time results. Examples of the applications of in-database analytics include transaction fraud detection, product recommendations, and web advertisement selection tailored for a particular user.

Entities, fields, and records

Every relational database contains one or more tables, each one of which contains data about a specific kind of thing, or ***entity***.

An entity is a type of thing, not a thing itself. For example: suppose we had a database storing information about our model vehicle business. In our model vehicle dealership, we have lots of different products, and we need to store important information about each product. To do this, we will build a table for the general ‘product’ entity, and we will use this table to collect information about each of our different products. As a result, the ‘product’ entity will be represented by the product table, and each specific product will be represented by a row of that table.

When we create our product table, we will list several things that we want to know about each specific product - the name, the category (for example, model motorcycle or boat), the quantity in stock, the price, *etc*. These are referred to as our ***fields***. In the table, they will be the columns. Another word for "field" is ***attribute***, because each column of a table represents a feature of the entity represented by that table.

As we add products, we will create a new row to store all the information about each one. This row is a ***record***. For each record, we will fill out all the fields. The table below contains brief definitions of each of the above terms.

Term	**Definition**
Entity	A type of thing
Record	An example of an entity

Field	An attribute of a record

Since a database is just a collection of tables combined with an account of how those tables relate to one another, we can see that each database provides a certain ***worldview*** that states which types of things exist and how they relate to one another. We might imagine different disciplines - from history, to biology, chemistry, physics, philosophy, economics, finance and business - each having its own database, each of which recognises (or "postulates") different entities or tables.

Relational algebra *(Optional section)*

This section is optional, so feel free to skip over it to the next one if you're not mathematically inclined. There's absolutely no shame in that!

If you do like maths, you can see a table as a ***mathematical relation*** (i.e, a set of tuples). As a reminder, a ***tuple*** is either a single object (e.g, me) or a pair (e.g, me and my father) or a triple (e.g, me, my father and my father's father) or a quadruple, *etc*. Single objects are called one-tuples, pairs are called two-tuples, triples three-tuples, *etc*. A one-place relation is simply a set of one-tuples (such as the set of all humans), a two-place relation (that is, a binary relation) is a set of two-tuples (such as the set of all pairs, the second member of which is the father of the first), a three-place relation is a set of three-tuples (such as the set of all triples, the third member of which is the father of the second member, who is the father of the first), *etc*. With this in mind, we can analyse a table with one column as just a one-place relation, a table with two columns as a two-place relation, and a table with *n* columns is an *n*-place relation.

Seeing tables this way, we might understand some SQL operations like JOIN, to which we'll soon be introduced, a little more precisely in terms of set-theoretical operations like *union* and *intersection*. But seeing SQL in this way is *not necessary* for being an excellent programmer, SQL-user or data scientist, so don't worry, and plough on, if these ideas don't resonate with you.

Data Types

Every column has a data type assigned to it. This is a requirement and feature of RDBMS which improves data quality by reducing accidental introduction of, for example, text into a numerical field. It is also a restriction if the column changes in the future - for example, if an account number is adapted to start with alphanumerical text several years into the operation of the system.

The data types in SQL are much like those in R and Python, but they have different names. While there are standard SQL data types, every RDBMS will have it's own complete set. Check out this table of the common data types in SQL:

Data Type	Explanation	Example

char(n)	A fixed length string of alphanumeric text, that must be of length n	`area_code CHAR(5)`
		`07837`
		`01653`

varchar(n)	A variable length string of alphanumeric text that can be any length up to a maximum length n	`email_address VARCHAR(100)` `myemail@hotmail.com`

int	An integer (whole number)	`view_count INT`
		`1`
		`195`

float	A number with floating point precision	`weight FLOAT`
		`0.18102010`
		`13.1213121`

boolean	A boolean true/false value	`married BOOLEAN`
		`TRUE`
		`FALSE`

date	A date	`date_of_birth DATE` `1990-03-04`
time	A time	`time_of_birth TIME` `01:00:01`

timestamp	A date and time	`date_of_birth TIMESTAMP`
		`1990-03-04`
		`01:00:01`

Primary keys

A table also needs an identifier field - a field containing only unique values, so that records can be differentiated from each other. It's possible (albeit unlikely) that two products might coincidentally end up with the same name. Let's go back to our model vehicle dealership database to exemplify this. '1968 Dodge Charger' could be of a 1:18 scale, and '1968 Dodge Charger' could be of a 1:1 scale; making them two products with the same name. As a result, we need to have some way of definitively separating similar records to avoid data corruption and confusion.

When you create an entity table, you will define a ***primary key*** - generally an ID field - that should *always* be *unique, non-null and unchangeable*. Perhaps one product has a primary key value of 'S18_1984', while the other has the primary key of 'S18_3233'. This means that you are free to have names and other fields that overlap without causing confusion. Let's check out an example:

productID	productName	productLine	quantityInStock	price
S10_1678	1969 Harley Davidson Ultimate Chopper	Motorcycles	7933	48.81
S10_1949	1952 Alpine Renault 1300	Classic Cars	7305	98.58
S10_2016	1996 Moto Guzzi 1100i	Motorcycles	6625	68.99
S10_4698	2003 Harley-Davidson Eagle Drag Bike	Motorcycles	2000	91.02
S10_4757	1972 Alfa Romeo GTA	Classic Cars	3252	85.68

This is the *products* table, and it describes an entity: the products of our database. Each row is a record or specific example of that entity, and each column is a field or attribute. Provided there are no NAs or null values, each record has a determinate quality, written within a cell, for each column. When creating a table, we need to explicitly define the table, its columns, data types, keys and any additional constraints before we can start adding in rows of data.

We could've created the above *products* table with a piece of SQL like this:

```
CREATE TABLE products (
  productID int NOT NULL,
  productName varchar(70) NOT NULL,
  productLine varchar(50) NOT NULL,
  quantityInStock int NOT NULL,
  price decimal(10,2) NOT NULL,
  PRIMARY KEY (productID)
);
```

What exactly does this code do? First off, the code makes a table, gives each column a name, and defines the type of each column. The 'NOT NULL' part of each line ensures that the field cannot be empty for any row. The primary key is marked as the productID field.

This is SQL being used as a ***DDL*** (or Data-Definition Language), as we're using SQL to make a data structure in which we will go on to store some data. SQL can also be used as a ***DML*** (or Data Manipulation Language); this is when we update, retrieve or delete information in our data structures with SQL queries. For example, the following query:

```
SELECT *
FROM products;
```

Is the 'Hello world!' of SQL queries, and simply returns all columns and all rows from the *products* table. You'll learn a lot more about how to construct SQL queries in the next subunit.

NoSQL

You might have heard of 'NoSQL,' but not quite know what it is. There's a lot of hype around this term!

It stands for 'Not Only SQL' (we know - it's a fairly misleading name!) and it's actually just an alternate way of designing a database to the relational database. To illustrate the difference, let's take an example; one that might come straight out of real life.

Suppose we're tracking medical information of patients in a big hospital in a large relational database. We have a *patients* table with the fields: *ID*, *Test1, Test2, Test3, DOB* and *BloodType*. Our *patients* table, however, is very ***sparse;*** that is, it contains lots of of null values. Not all patients have taken all three tests, in fact, most patients have only taken one of the three tests (in the cell-values are recorded the relevant patient's value for that test). In addition, not all of the patients have a known blood type recorded in the table. Suppose our table looks like this:

ID	TEST1	TEST2	TEST3	DOB	BTYPE
				1-06	
2732	80		95	1965	A-
				15 07	
2946		92		1965	
				16 07	
3650	86			1965	O

It's a shame, but many relational databases contain tables that look just like this! Notice the poor ***data integrity***; the values for the DOB column spill out over two rows: patient 2732's DOB, for example, is actually 1-06-1965, but the month and day are on the row above the one containing

the year! Let's imagine this table is also just *enormous*; we are in the realm of ***big data***, and we have a big, unwieldy relational database containing super sparse tables just like this one.

In such a situation, we might prefer to arrange our data in a key-value table, as follows:

ID	KEY	VALUE
2732	TEST1	80
2732	TEST3	95
2732	DOB	1-06-1965
2732	BTYPE	A-
2946	TEST2	92
2946	DOB	15-07-1965
3650	TEST1	86
3650	DOB	16-07-1965
3650	BTYPE	O

Notice how we now have a table with no null values at all! Sure, each row now contains a different sort of information about the patient, but we've replaced our confusing sparse table with a long set of identifiable key-value pairs adequate to the task of storing extremely large stocks of data on our patients. That patient 2946 has no known blood type in our system doesn't put a null value in our table anymore.

NoSQL works precisely by replacing complex relational databases, in which the tables relate to one another in interesting and neat ways, by big three-column tables of the sort exemplified above. These tables are said to use the ***key-value*** (or ***associative array)*** data model, and such tables are capable of storing very large quantities of sparse data, where the number of attributes is very large, but the number of instances of those attributes is relatively low. With NoSQL, *there is no schema at all,* which means that the relationships between the kinds of things described by our data can be violated by mistaken data entry.

As ever with technologies, NoSQL has its merits and demerits. We can capture these in the following table:

Advantages of NoSQL	Disadvantages of NoSQL
Can make sense of big data, stored in large, sparse tables in relational databases.	Very primitive data model. The relationships between the entities described by the data are obscure to the users and have to be painstakingly reconstructed when required.
High performance and scalability. Data can be retrieved, deleted and updated quickly; even with enormous datasets.	Transaction management is negligible, due to there being no management system.
Fast updates: to add a new entity attribute, we just add a new row.	Mistakes can be made easily and these are hard to track.
Easily supports ***distributed*** data architectures (architectures where the data is stored in different physical locations).	The application program manages the validation of data and the integrity constraints. As a result, if this crashes, the data can be corrupted.

Wrapping Up

We hope you've enjoyed this introductory article on databases, DBMS, SQL and NoSQL. As we've seen, there's quite a lot to them, and they're actually extremely interesting! Don't worry if you didn't grasp absolutely everything, though: in the rest of the sub-unit, you'll check out a few resources to cement what you've learned. These will include a short video on relational databases, an article on which databases are most popular in industry, and a great piece on the set theoretical underpinnings for SQL and RDBMS. Enjoy, and good luck!