Springboard

Data Science Career Track
Introduction to SQL and Databases

Databases, DBMS, and SQL

The best solution for storing large amounts of complex data is through the use of a ***database***. In this article, we're going to introduce you to key concepts of this technology, a foundation of our high tech world.

On a basic level, a ***database*** is an organized collection of structured information collection of virtual tables that store information. This is often implemented as a collection of virtual tables that store information, combined with an account of how those tables relate to one another. Generally, when we use the word "database" we're referring to a type of software called a Database Management System or ***DBMS***. Database Management Systems run on a dedicated machine - a server - or run on your own machine - a local instance.

There are many different DBMS: *MySQL*, *Microsoft SQL Server,* and *PostgreSQL* to name a few. There's no 'best' DBMS. Each has its own merits and downsides. Check out this optional link for a comparison of popular DBMS.

Relational Databases, popularized by Oracle Corporation, are the standard database technology of the working world - 90% of Fortune 500 companies use Relational Databases in their daily operations. Relational Databases store data in a series of tables with links connecting data from one table with data in another. Because of their prevalence and ease of use, we'll be using this technology to teach you about SQL and Data Science. The underlying theory behind Relational

Databases is rooted in relational algebra and set theory, we'll explore this in a video later in the unit.

SQL 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 easy to use syntax. SQL is especially forgiving in its rules, making it a sweetly satisfying and immediately applicable language for the budding data scientist's arsenal. The exact SQL syntax varies depending on the DBMS; what you write to query a MySQL database will be slightly different from PostgreSQL. However, the syntax is similar enough that once you learn one, you should be able to read the others, no matter the DBMS.

Entities, Fields, and Records

Every relational database comprises of one or more tables that contain data about a specific ***entity***.

An entity is a type of thing, not a thing itself. An entity would be the general class of things: Company, Tree, or State. An entity would not be Oracle, Redwood, or Virginia.

For this example: suppose we are running a company that makes model vehicles like miniature Ferraris and Harley Davidsons. We want to store information about the different models we have in stock, so we build a general 'Model' entity table. That's where we'll store all the **records** of different models, and all the important **attributes** these models have. The entity is generally model vehicle, the record is for a specific model, and the fields are the qualities these specific models have.

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. If there are no null values, each record has valid data within each cell for each column.

productID	**productName**	**productLine**	**quantityInStock**	**price**
S10_1678	1969 Harley Davidson	Motorcycles	7933	48.81

	Ultimate Chopper			
S10_1949	1952 Alpine Renault 1300	Classic Cars	7305	98.58
S10_2016	1996 Moto Guzzi 1100i	Motorcycles	6625	68.99

For each model vehicle we sell, we'll create new **records,** rows to store information **fields** about our model vehicles: what the name is, which type it is, how many we have in stock, etc. The table below contains brief definitions of each of the above terms.

When we create our product table, we'll list ***fields*****.** Fields are properties 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*. In the table, they will be the columns. Fields are also called ***attributes*** because each table column represents a feature of the entity.

Term	**Definition**
Entity	A type of thing
Record	An example of an entity
Field	An attribute of a record

Data Types

Every column has an associated data type. This feature of RDBMS improves data quality by preventing the accidental introduction of, for example, text into a numerical field. The advantage of this feature is that it improves data quality by ensuring all data is the same type. A weakness of giving a column an associated data type is that it is restrictive. For example, look at our productID from the model vehicle sample table. Let's say a new manager decides to remove the prefix 'S10_' from the productID column, in order to make productID an int instead of a varchar, there is no way to change the datatype of the column. They would have to create an entirely new column of a new data type.

SQL has standard data types, but every DBMS has its own implementation of these data types. Even if the names are the same, the data types can vary on details like storage size.

Here are 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

Records require an identifier field - containing a unique value so we can differentiate records. It's possible that two rows end up with the same information - like two ‘John Smiths’ in a staff directory. We need a way of differentiating these records to avoid data corruption and confusion.

Let’s look at our model vehicle dealership database. Say we have two products with the same name: a ‘1968 Dodge Charger’ with a 1:18 scale and another ‘1968 Dodge Charger’ with a 1:1 scale. A unique identity field will work to separate these records.

When you create an entity table, you will define a ***primary key*** that should *always* be *unique, non-null, and immutable*. This means the primary key will never have a duplicate, it will contain real data, and no one will be able to change it. Let's say our "1969 Harley Davidson Ultimate Chopper" comes in two scales, 1:1 and 1:18. In an unlikely scenario, both these models have the same price and stock, and we don't have a column that keeps track of scale. A primary key will allow us to differentiate between these products when the fields to overlap. Let's check out an example:

productID	**productName**	**productLine**	**quantityInStock**	**price**
S10_1678	1969 Harley Davidson Ultimate Chopper	Motorcycles	2000	91.02
S10_4698	1969 Harley Davidson Ultimate Chopper	Motorcycles	2000	91.02

When creating a table, we need to explicitly define the table, its columns, data types, keys, and any additional constraints before we can start work with it.

We could've created the above 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)
);
```

This code creates a table, gives each column a name, and defines the data type of each column. 'NOT NULL' is a constraint that says each field must contain non-empty data. The primary key is marked as the productID field.

This SQL is used as a ***DDL*** (Data-Definition Language), as we're using SQL to make a data structure. SQL can also be used as a ***DML*** (Data Manipulation Language) when we update, retrieve, or delete information in our data structures with SQL queries.

This query is the 'Hello World!' of SQL:

```
SELECT *
FROM products;
```

It returns all columns and rows from the *products* table. You'll learn more about how to construct SQL queries in the next subunit.

A Brief Look At NoSQL

NoSQL, which stands for 'Not Only SQL,' refers to databases that are not structured relationally. Let's take a look at an illuminating example.

Suppose we're a hospital tracking medical information with a large relational database. We have a *patients* table with the fields: *ID*, *Test1, Test2, Test3, DOB,* and *BloodType*. Not all patients take all three tests: in fact, most patients have only take one of the three tests. Also, not all of the patients have their blood type recorded. Because of this, our *patients* table is ***sparse:*** it contains many null values. Suppose it 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 just like this! Notice the poor ***data integrity***: the values for the DOB column spill out over two rows, with the year on one row and the day and month below it! Patient 2732's DOB, for example, is actually 1-06-1965.

In many cases, we don't want a big, unwieldy relational database full of super sparse tables like this one. We'd 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! We've replaced our confusing, sparse table with a long set of identifiable key-value pairs. Analytically, a table like this is better for storing extremely large datasets about our patients than a sparse relational table. Though we don't know patient 2946's blood type, we don't need to put a NULL value into our table anymore.

NoSQL works by replacing complex relational databases, in which the tables relate to one another in ways we define, by three-column tables of the sort exemplified above. These tables use the ***key-value*** (or ***associative array)*** data model. Such tables are great for storing enormous 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's no schema at all, which means that our data is susceptible to bad data entry.

NoSQL has its strengths and weaknesses:

Advantages of NoSQL	**Disadvantages of NoSQL**
Can make sense of big data, stored in large, sparse tables in relational databases.	Simplistic data model. The relationships between the entities described by the data are obscured and may need painstaking user reconstruction.
High performance and scalability. Data can be retrieved, deleted, and updated quickly, even with enormous datasets.	Transaction management can be difficult depending on the DBMS
Fast updates: to add a new entity attribute, just a new row.	Mistakes are easily made and hard to track.
Easily supports ***distributed*** data	The application program manages the

architectures, where the data is stored in different physical locations	validation of data and integrity constraints. If it crashes, the data can be corrupted.

Wrapping Up

We hope you've enjoyed this introductory article on databases, DBMS, SQL, and NoSQL. There's quite a lot to learn about this interesting material! Don't worry if you didn't grasp absolutely everything. In the rest of the sub-unit, you'll go through resources to cement your knowledge. Enjoy, and good luck!