1 SQL and NoSQL

Structured Query Language (SQL[1]) is the programming language used across all industries and research, to create relational databases. A 'database' is a collection of information about anything, and the way this information is stored and accessed is via the informational 'relations'. Practically, we refer to different 'tables' when organising data. When constructing a database, one typically defines different tables for logically distinct bits of information. Each table will have a set number of rows (cardinality) and number of columns (degree). Formally, each row is separately called a tuple and each header is called an attribute; however, don't worry about memorising these terms (cardinality, degree, tuple, attribute). For example, a database for a shop, which would be accessed through the website, may have a table for: users, passwords, orders, email addresses, etc.

Ethics: Why might we keep passwords or emails addresses in a separate table, and how do you think we reference that table?

All tables in SQL need to have their 'schema' defined at the moment of creation. This means, for each attribute (column), one needs to know the type of data that will be entered. The type of data cannot be different for any row within that column, this is 'homogeneous' data. Depending on the exact flavour of SQL you are using, there are dozens of types of data. We focus here on the main forms you will typically encounter, although be aware that the exact names can change between software.

- INTEGER: whole numbers (e.g. 10).
- REAL: decimal values (e.g. 3.14).
- TEXT: strings (e.g. "hello world").
- BLOB: a binary object, typically things like images.
- BOOLEAN: a TRUE/ FALSE flag.

Then, a developer will create further restrictions on each attribute, known as 'CONSTRAINTS'. Important CONSTRAINTS include:

- NOT NULL: no value within the column can have missing data (NULL[2]).
- UNIQUE: a requirement that no values in a column can be repeated.
- AUTO_INCREMENT: automatically add 1 to a value when new records are added to a table.

[1] pronounced either "S-Q-L" or "sequel"

[2] It is important to recognise the distinction between NULL and 0 or FALSE. NULL represents where there is missing data.

- PRIMARY KEY: indicating this column indexes the table (makes it easily searchable). There can only be **one** PRIMARY KEY in a table and it also needs to be NOT NULL and UNIQUE.
- FOREIGN KEY: this links a table with the PRIMARY KEY elsewhere (this can be the same table, "self referencing", but is most often another table, often the "main" table).

Consider: Why do you think we use a FOREIGN KEY? What happens to records in sub-tables using a FOREIGN KEY when the main table deletes that KEY?

The reason for the "strictness" of SQL is historical. Data storage was a challenge and limiting factor in computer science until around 2010. Because of this, SQL was designed in such a way to be as efficient as possible, to keep down data storage costs. Examples include requiring certain data types to manually declare how many characters or bytes are required. Additionally, when each hard drive was extremely limited in volume, it is very beneficial to separate tables onto different hard drives (logically partitioning the data). This partitioning was an early form of 'horizontal scaling'[3]; by comparison, 'vertical scaling' (building bigger, more powerful servers) is much more expensive. As such, whilst the data storage became much cheaper, other data models than SQL were developed. These are collectively known as "NoSQL" ("not only SQL" or "non-relational SQL"). NoSQL data models tend to be faster to develop and are easier to scale horizontally through 'sharding' (using many different machines to spread the database load). The simplest NoSQL data model is "key-value", as in a `python` dictionary, where there is simply a key with an associated value, e.g. "username"="admin". Another very convenient NoSQL data model is that of the 'document store', where many searchable files (JSON, XML, etc.) are stored together. The document store is incredibly flexible, which can be either a benefit or disadvantage, depending on the use case.

Consider: Think about different use cases for the different data models we have discussed here.

1.1 Constructing a query

Querying a database has a simple base form, from which you can expand into more complicated queries. SQL is case-insensitive but the standard nomenclature is to use SQL commands in all-caps, anything you see in brackets is optional. It is expected that queries are ended with a semi-colon, ';', but many SQL interfaces will do this for you.

Columns are accessed as <TableName>.<columnName> (note the use of upper and lower camelCase). SQL comments are created with '--'. **Always**

[3] "scaling" is typically the case for living databases, where more records are constantly being added.

check the documentation for the specific commands in the software you are using.

SELECT <attribute>
FROM <TableName> [**AS** <Name>]
WHERE [**NOT**] <condition>;

Here, we SELECT an attribute, where '*' is a wildcard which translates to "all columns"; FROM the table we want to query, with the optional AS to rename a table temporarily; WHERE a Boolean condition is TRUE (or FALSE if using the NOT optional). One can also chain WHERE expressions with AND or OR. A comma, ',', is used when selecting multiple attributes (e.g. SELECT <column1>, <column2>, <column3>). We can also modify these queries with further SQL commands.

1. SELECT:
    - DISTINCT <attribute>: Ensure only unique values are returned from a given column.
    - COUNT(<attribute>): Returns only one row with the number corresponding to that query.
    - AVG(<attribute>): Returns only one row with the mean average for the given numeric column.
    - SUM(<attribute>): Returns only one row with the sum for the given numeric column.
    - MIN(<attribute>): Returns only one row with the minimum value from the given numeric column.
    - MAX(<attribute>): Returns only one row with the maximum value from the given numeric column.
2. WHERE:
    - <x> BETWEEN <a> AND <b>: A shorthand conditional for numeric columns equal to $a \geq x \geq b$.
    - <x> LIKE "%<y>_": When checking a string you can compare to another sub-string; '%' is a wildcard searching for any length sub-string whilst '_' allows any, singular character. For example, on a list of the UK countries, LIKE "%lan_", would return England, Scotland, and Northern Ireland, but not Wales.
    - IN (<a>, <b>, <c>, ...): Can be used for either numeric values or strings and allows one to check for multiple values exactly matching, at once.

You can then filter your results with the following optional commands:

[**GROUP BY** <attribute> [**ASC**|**DESC**]]
[**ORDER BY** <attribute> [**ASC**|**DESC**]]
[**LIMIT** N];

- GROUP BY: This allows you to group the results by the given column in either ascending (ASC, default) or descending (DESC) order. A typical use case is to pair this with a mathematical function like AVG or SUM.
- ORDER BY: Similar to GROUP, this displays the results in either ascending or descending order.
- LIMIT N: Limit the number of results returned to N amount.

Finally, a common task is to join multiple tables together with the JOIN command:

```
SELECT <attribute>
FROM <TableName> [AS <Name>]
JOIN <TableName2> [AS <Name2>]
USING (<sharedColumn>)
WHERE [NOT] <condition>;
```

or:

```
SELECT <attribute>
FROM <TableName> [AS <Name>]
JOIN <TableName2> [AS <Name2>]
ON <TableName>.<columnName> = <TableName2>.<columnName2>
WHERE [NOT] <condition>;
```

The different kinds of joins are:

- (INNER) JOIN: The default JOIN of tables, returning just the rows which are shared by both tables.
- LEFT (OUTER) JOIN: All rows from the first table, with second table matches included if present.
- RIGHT (OUTER) JOIN: All rows from the second table, with first table matches included if present.
- FULL (OUTER) JOIN: Returns all matches if found in either table.