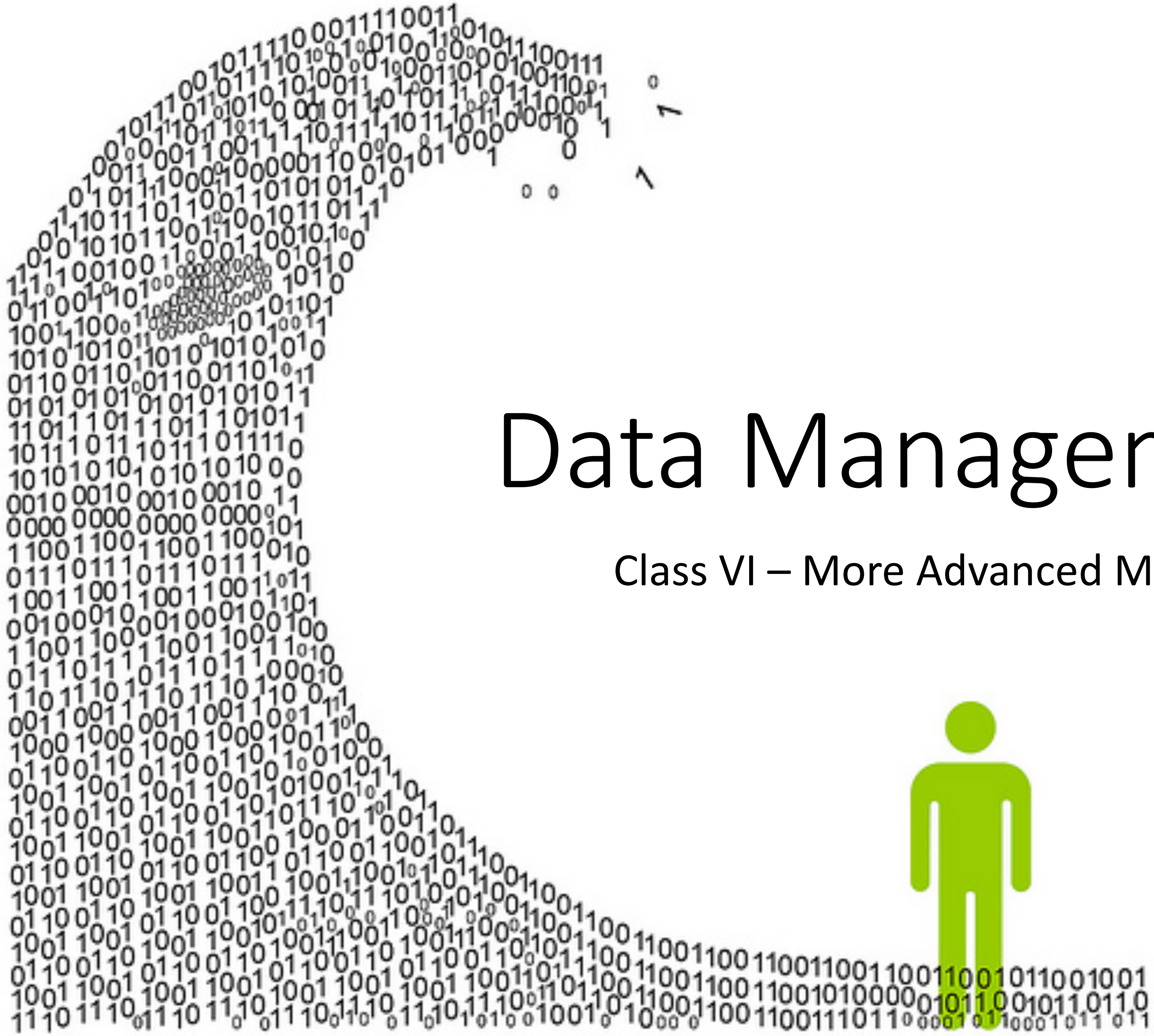


Data Management

Class VI – More Advanced MySQL

Revision

- In the last class, we (hopefully!) got to the point where you could:
  - Design a MySQL table for your data (CREATE TABLE);
  - Insert your data into the table (INSERT);
  - Write queries to filter, sort and extract data (SELECT).
- These are the basics of any SQL system, from a database on your laptop to a gigantic commercial data set in the cloud.

Today’s Objective

- So far, what we’ve done with SQL is really just a copy of what you can do with data frames in R or Python.
    - You might reasonably be thinking, “why am I learning a *third* way to do basically the same things…?”
- Today we’re going to talk about three major things you can do in SQL that generally aren’t possible in other software:
    1. **Streaming data**
    2. **Indexing data**
    3. **Querying across more than one table**

About Skype Classes…

- Because we’re doing these classes on Skype, which was not the original plan, there are some limits on the kind of in-class exercises we can do – I can’t easily walk around and look at all the code you’re all writing.
- We’ll have to figure out a format for these classes that works for all of us, but for now, it’s possible that the classes will be slightly shorter than usual, in order to leave time for more one-to-one Skype or email consultations where I can see the work you’re doing.

① Streaming Data

- One of the major reasons for using a database is to handle very big tables of data – perhaps too large for Python or R to deal with on their own.
- However, so far we’ve been moving data from SQL into Python just by copying it into a data frame – using the **cursor.fetchall()** command, which gets *all* of the data from our query.
- It’s still useful to be able to filter data, but if ultimately we’re copying it all into Python or R, there’s no advantage if the data didn’t fit in the first place.

Streaming Data (2)

- This is where *streaming*, rather than *copying*, information from the database becomes important. You can think of it in exactly the same way as “streaming” a video from Netflix or Youtube – you don’t have to download and save the entire video file in order to watch it, right?
- We can do the same thing with data. Many kinds of analysis actually only need to work on one piece of data at a time; for example, when we tokenise text or turn it into a vector (term-document matrix), we actually only work on one document at a time.

Streaming Data (3)

- To do this, instead of **cursor.fetchall()** we use **cursor.fetchone()**, which just gets one row at a time from the database.
- By writing a loop in Python or R, you can get a single row of data, do whatever processing you need (tokenisation, vectorisation etc.) and then move on to the next row. *You only need to store one row of data at a time in memory*.
- This approach is a little slower than storing everything in memory - but it allows you to work with data so big that Python or R would just give you an error if you tried to load it all.

Streaming Data (Example)

```
cur.execute(“SELECT * FROM big_data_table”)
while True:
    one_record = cur.fetchone()
    if one_record is None:
        break
    do_some_processing(one_record)
```

Streaming Data (Example Notes)

- The **while** command creates a loop which will keep going until the condition is False.
  - Normally you'd write something like **`while x < 6:`** which would keep going until the variable x became six or larger. In this case, we're creating an *infinite loop* – the condition is always True. It will only stop when our program uses the **break** command.
- The **break** command breaks out of any loop (**while, for** etc.) and moves on to the next section of code outside the loop.
- When we reach the last record in our database, running **`cur.fetchone()`** again will return "None" – so we can test for this to know when it's time to use **break** and finish the loop.

Streaming Data (Final Thoughts)

- You don’t need to use data streaming when the data is small enough to just load into a data frame. Remember one of our first principles – don’t do something complicated when there’s a simple option available to you.
- Often, analysing data requires going over the same data set multiple times – tokenisation followed by vectorisation, for example, or training epochs of a classifier or topic model. That’s fine! Don’t be afraid to stream data in and out of the database multiple times. The database doesn’t get tired and hasn’t joined a trade union.

② Indexing Data

- When you have a very big set of data, searching that data can start to take a long time. No matter how powerful the database software is, or how much memory and CPU power the PC it’s running on is, searching through gigabytes of data still takes time.
- You can speed things up enormously, however, by creating an **index** – telling the database that certain columns are the ones you’re most likely to search for, so it can quietly create a quick look-up table in the background, and return search results much faster.

Indexing Data (2)

- There are two ways to create an index:
  - You can tell SQL what you want your indexes to be when you make the table for the first time, using the CREATE TABLE statement. This allows the database to update the indexes automatically every time you add a new piece of data.
  - Alternatively, you can define an index later on, using the ALTER TABLE statement. This forces the database to create a new index for all the data already in the table – if there's a lot of data there, this can take a long time.
  - Since we already made some data tables, we'll focus on the ALTER TABLE command.
- Whichever approach you use, actually *using* the index is transparent – you just use normal SELECT commands, and where possible the database will use your index to speed things up.

How to choose your Index

- You should create indices for columns you’re going to use in your search very often.
- For example, imagine you have a gigantic table of tweets, and you’re interested in analysing tweets sent by individual users. That means you’ll very often search on **screen_name** (or **user_id**) to get all the tweets from an individual user.
- Normally, the database has to examine every tweet to see if the screen name matches your target user. But if you create an index for the **screen_name** column, it can quickly locate all those entries without doing a full search of all the data.

Creating an index

- Here’s how to add a simple index to an existing table:

```
ALTER TABLE my_table ADD INDEX (column_name)
```

- You can also create an index from *multiple* columns. Let’s say you know that you’re going to do a lot of searches that simultaneously use both the **name** column and the **city** column:

```
ALTER TABLE my_table ADD INDEX (name, city)
```

Unique and Fulltext Indices

- You can also define a UNIQUE index. The value in this column can only appear once in the the database – try to put in a row with the same value again, and it'll cause an error. This is useful when you're storing data that shouldn't appear multiple times (like Twitter usernames).

```
ALTER TABLE my_table ADD UNIQUE INDEX (twitter_id)
```

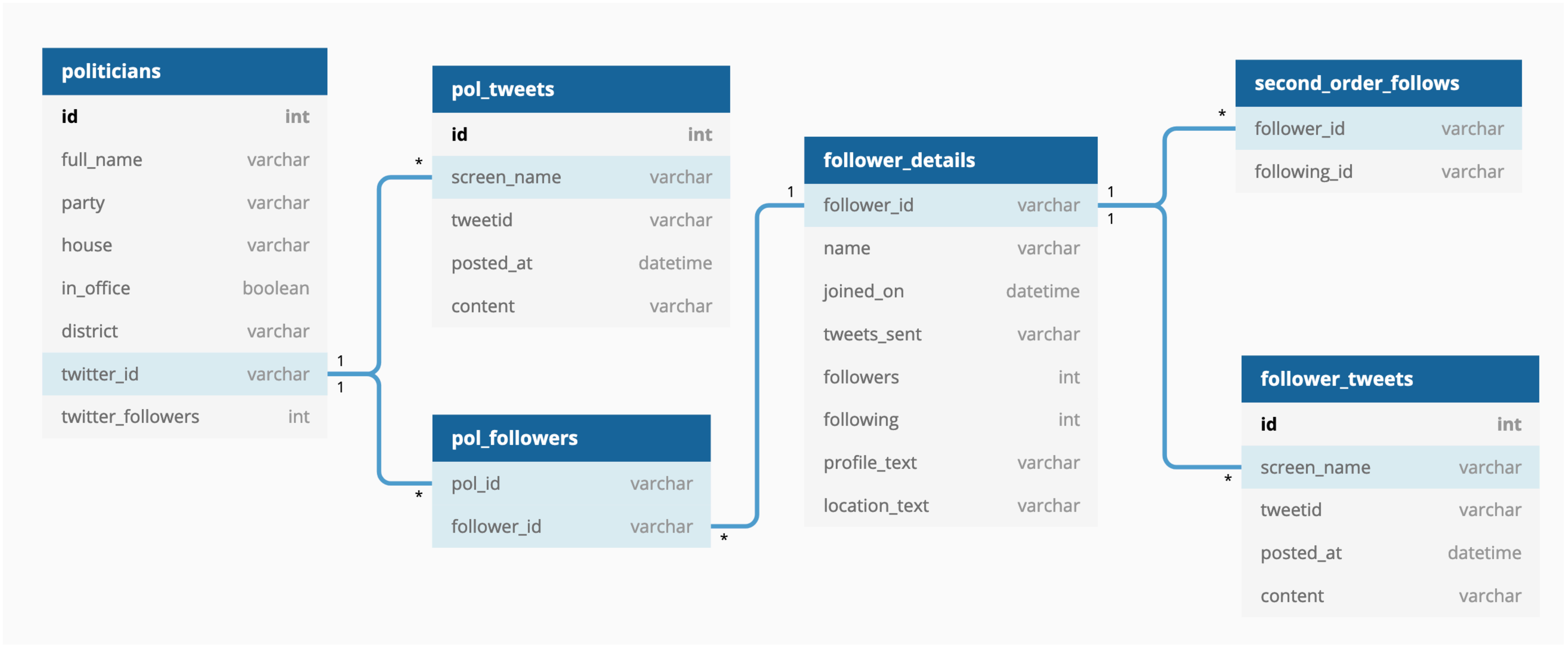
- Another kind of index is FULLTEXT. This is usually set on a TEXT column and allows you to search inside the text quickly. FULLTEXT indices take some time to create, especially on a large database, but they massively speed up searching.

```
ALTER TABLE my_table ADD FULLTEXT INDEX (article_text)
```

③ Querying Multiple Tables

- The final and perhaps most powerful unique feature of SQL compared to data frames that we'll talk about today is the **JOIN** command, which lets you search and combine data from multiple different tables.
- There are many cases where this is useful; often, you'll want to create a database storing different kinds of data (information about a user, information about the people they follow on social media, information about their social media posts, etc.) which is linked together by *keys,* unique identifiers like Twitter IDs that are found in all the different bits of data.

A set of linked tables...



politicians	
id	**int**
full_name	varchar
party	varchar
house	varchar
in_office	boolean
district	varchar
twitter_id	varchar
twitter_followers	int

pol_tweets	
id	**int**
screen_name	varchar
tweetid	varchar
posted_at	datetime
content	varchar

pol_followers	
pol_id	varchar
follower_id	varchar

follower_details	
follower_id	varchar
name	varchar
joined_on	datetime
tweets_sent	varchar
followers	int
following	int
profile_text	varchar
location_text	varchar

second_order_follows	
follower_id	varchar
following_id	varchar

follower_tweets	
id	**int**
screen_name	varchar
tweetid	varchar
posted_at	datetime
content	varchar

Querying Multiple Tables (2)

- To write a query that uses data from multiple tables, we use the **JOIN** command. This literally “joins” the two tables together, so all their columns become part of one ‘super-table’ which we can filter and query using the usual tools in the **SELECT** command.
- Usually, you want to make sure that the correct rows in Table A are joined with the rows in Table B – for example, that a Twitter user in Table A is only joined up with their own tweets from Table B.
- This is accomplished using the **ON** statement, which tells SQL how to choose the rows to join.

Querying Multiple Tables (3)

- This is a very simple example of an SQL JOIN…

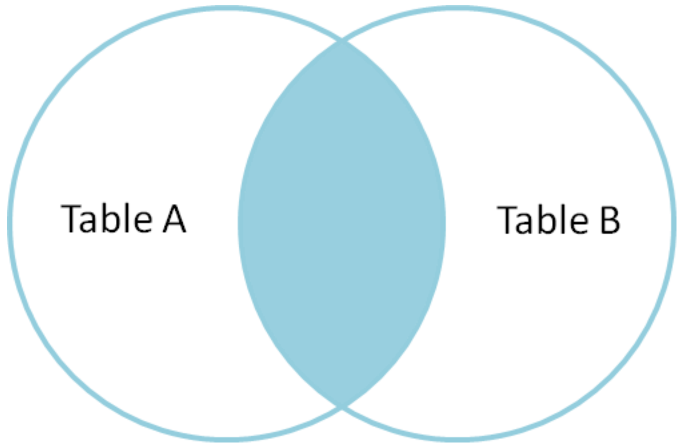
```
SELECT * FROM table1
INNER JOIN table2
ON table1.id = table2.id
```

- This query would return a single row with all of table1’s columns *and* all of table2’s columns, for every single row in either column where the ‘id’ value was the same.

Inner and Outer JOINs

- In the last example, we used an “INNER JOIN”. This is the most common kind of JOIN, and it returns only the records in Table A and Table B which have the same ID field.
  - In other words, if there’s data in either table which *doesn’t* have matching data in the other table – for example, a Twitter user in Table A who doesn’t have any tweets stored in Table B – their data won’t be returned at all.

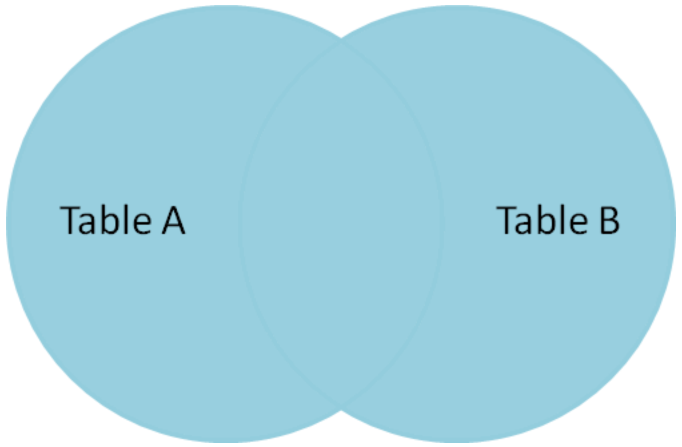
- It’s called an “inner” join because like the Venn diagram opposite, only records which *overlap* between the two tables are returned in the query.



Inner and Outer JOINs

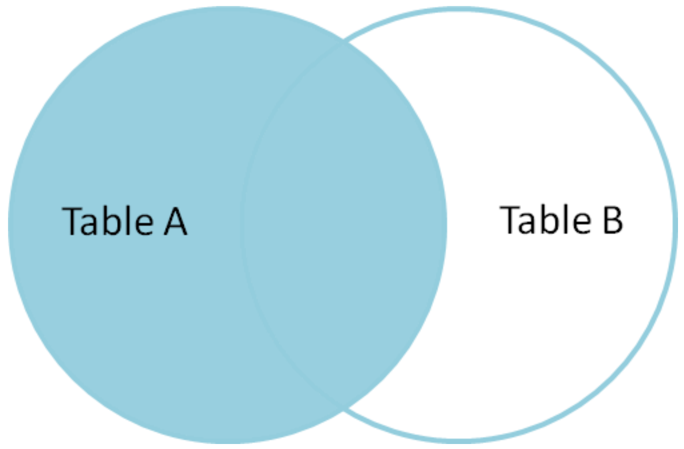
- There are also OUTER joins – the most commonly used of which are LEFT and RIGHT joins.
  - These will return records from tables *even when there isn't a matching record in the other table* – so the values in the columns from those tables will become "NULL", showing that there was no match.

- This is a FULL OUTER JOIN between Table A and Table B. Every record from both tables will be returned, with "NULL" values filling in when there wasn't a match in the other table.

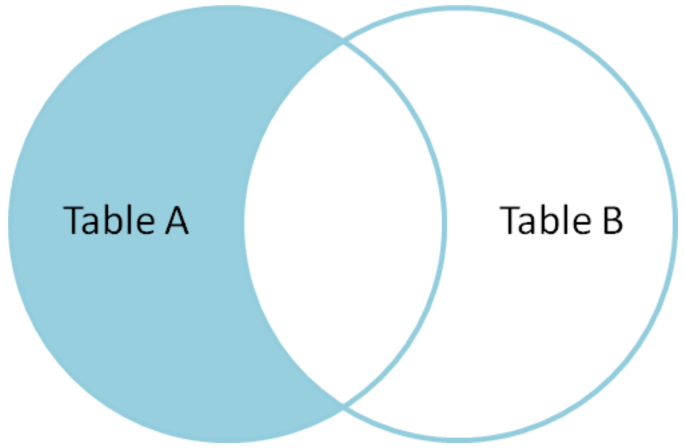


Inner and Outer JOINs (3)

- This is a LEFT JOIN between Table A and Table B. Every record from Table A is returned, with "NULL" values filling in when there wasn't a match in Table B.



- This is also a LEFT JOIN between Table A and Table B – but here, we check if the Table B columns are NULL (e.g. **WHERE tableB.id IS NULL**) and only return those columns. This way we can find which records in Table A have no matches in Table B.



More on JOINs

- Using the JOIN commands is really powerful, as it allows you to combine data from multiple tables easily. One common usage is to create a new table to store your analysis results – this way, JOIN lets you easily access the original data and analysis results together.
- It definitely takes a while to understand how these commands work. For now, the main thing is to know that they exist and are available to you… When you need to actually use them, you can always find detailed guides and references online.

SQL Best Practices

- Finally for today, I thought we should talk a little bit about how to most effectively use SQL databases.
- There are some things about SQL which are different to how you’re used to doing things, and these can be a bit counter-intuitive.

1) Storage is Cheap

- A lot of people have the instinct to try to minimise the amount of data they store – they avoid using too many tables, for example.
- In reality, **storage is cheap**. Even your smartphone probably has at least 64Gb of storage – enough to store vast amounts of text data. Cloud services only charge a few cents a month for storing gigabytes of data.
- So… Don't be afraid to store *lots* of tables, and in general, *never delete data*. Instead of deleting data from tables, make a column called "deleted" (or whatever you like) and set it to True when you don't want to use that data. You never know if it might be useful in future.

2) Processing Takes Time

- Storage is cheap… But processing always takes time, even on the fastest computer.
- Your objective should be to make sure that you never have to recalculate the same analysis result twice.
- So, when you run an analysis – especially one that takes time, like tokenisation – you can often benefit by making a new table and storing your analysis results.
- Next time, you just look up the result in the database, rather than having to run the analysis all over again!

3) Back up, back up, and back up some more.

- SQL databases do a lot of clever stuff to keep your data safe… But ultimately, if they're running on your laptop, they can't rescue you if you drop the laptop in a lake, or it gets stolen in a bar.
- MySQL provides a command line program called 'mysqldump' which will 'dump' your databases and tables out into a backup file – which you can then save on a network drive, an external hard disc etc.
- You can download tools to do this automatically every few days. It's a really, really good idea to do so!