

GENERAL ASSEMBLY

SQL BOOTCAMP

LET'S MEET



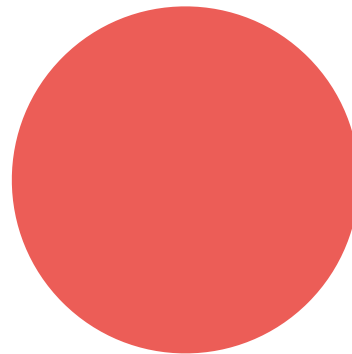
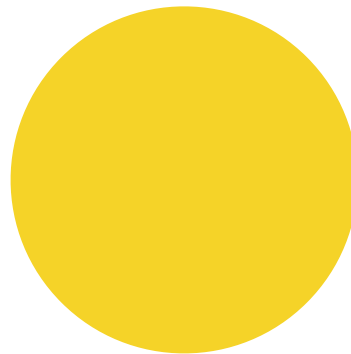
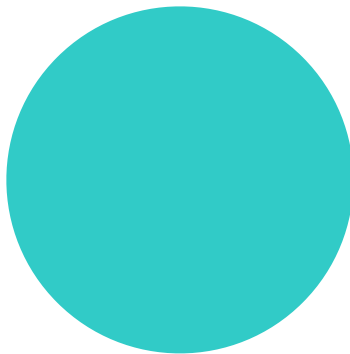
ABOUT YOUR PRODUCER!



Chris Wright

chris@generalassemb.ly

Classes & Workshops Lead



ABOUT YOUR INSTRUCTOR!



Ruben Naeff

rubennaeff@gmail.com

Data Science Instructor

**DATA SCIENTIST
AT KNEWTON**

**MUSIC COMPOSER
STRATEGY CONSULTANT
ECONOMIC RESEARCHER
MATH TEACHER**

**AMSTERDAM, NL
BROOKLYN, NY**

ABOUT YOU!



Who
are you

What
do you do

Why
are you here

LET'S SET UP



DOWNLOAD REPO TO MACHINE

- Go to https://github.com/rubennaeff/sql_bootcamp
- Click Download ZIP

OR

- `git clone https://github.com/rubennaeff/sql_bootcamp.git`

EVERYONE ALL SET WITH THE INSTALLATION?

- A MYSQL SERVER**
- A MYSQL CLIENT**

0. MEET, SETUP, TROUBLESHOOT – DONE!

I. INTRO TO DATABASES & BASIC SQL

II. AGGREGATIONS & GROUP BY

III. RELATIONAL DATABASES & JOIN

IV. CREATE DATABASES & TABLES

SQL BOOTCAMP

INTRO TO DATABASES

Before we start:

Where and how do you store your data?

Before we start:

Where and how do you store your data?

For example, I do my household budgeting in Google Slides.

More examples?

What is ETL?

- **E**xtract data
- **T**ransform data
- **L**oad data

Databases are a **structured** data source optimized for efficient **retrieval and storage**

Databases are a **structured** data source optimized for efficient **retrieval and storage**

structured: we'll have to define some pre-defined organization

Databases are a **structured** data source optimized for efficient **retrieval and storage**

structured: we'll have to define some pre-defined organization
e.g., a table with columns for first name, last name, DOB, address, etc.

Databases are a **structured** data source optimized for efficient **retrieval and storage**

structured: we'll have to define some pre-defined organization

retrieval: the ability to read data our

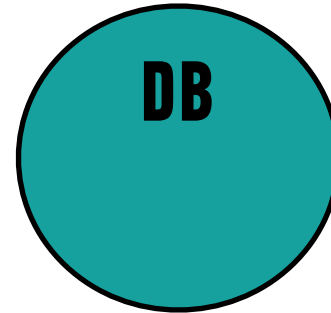
storage: the ability to write data and save it

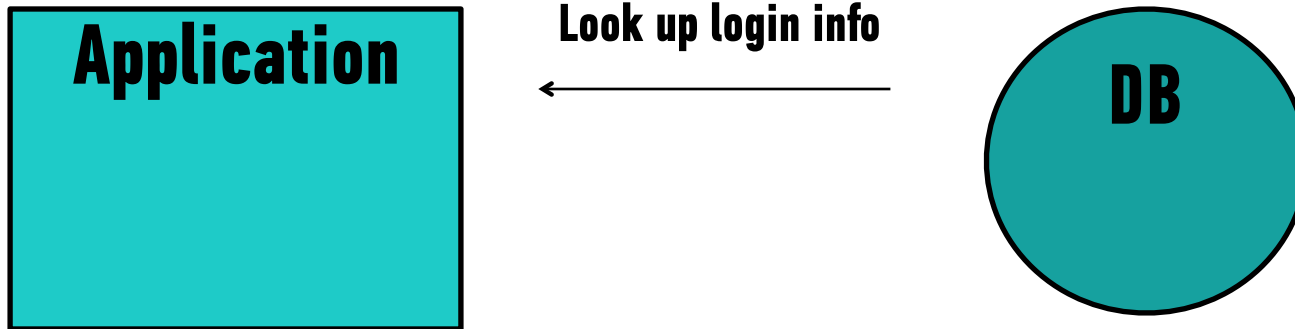
Databases are a **structured** data source optimized for efficient **retrieval and persistent storage**

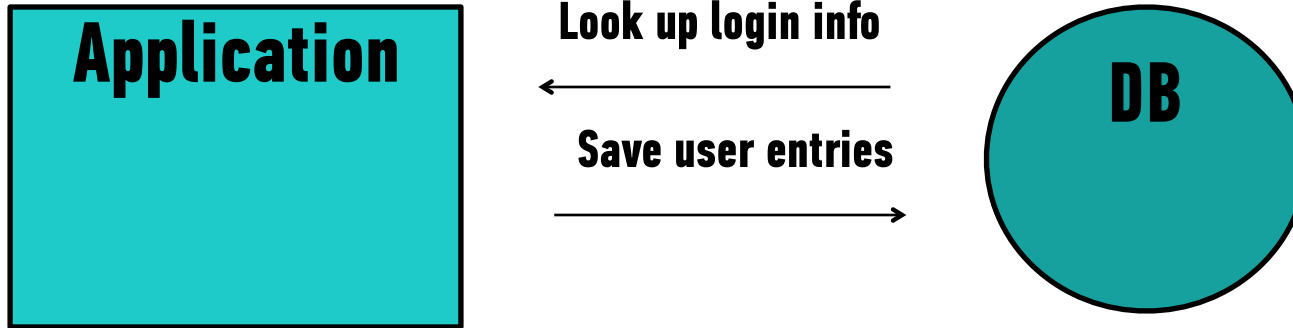
structured: we'll have to define some pre-defined organization

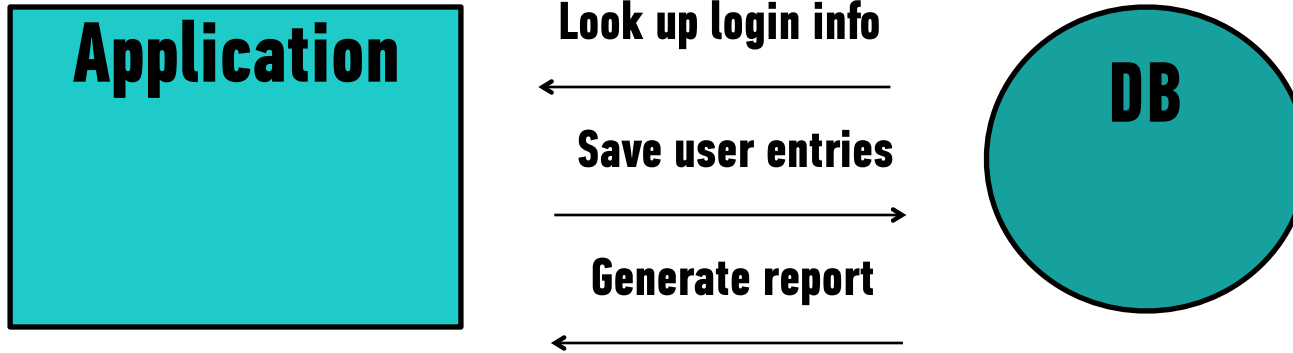
retrieval: the ability to read data our

storage: the ability to write data and save it









Thinking about the personal quiz question of earlier,
when is a database useful, and when other storage types?

SQL BOOTCAMP

SQL (STRUCTURED QUERY LANGUAGE)

SQL (Structured Query Language) is a query language designed to extract, transform and load data in relational databases

SELECT *
FROM table

SELECT *
FROM table

*Select all columns
from this table*

SELECT col1, col2
FROM table

*Select some columns
from this table*

SELECT DISTINCT col1, col2
FROM table

Only select unique entries.


```
SELECT *  
FROM table  
WHERE <condition>
```

SELECT *

FROM table

WHERE <condition>

SELECT customer, spend

FROM sales

WHERE spend > 100

SELECT *
FROM table
WHERE <condition>

SELECT customer, spend, city
FROM sales
WHERE spend > 100
AND city = 'NYC'

SELECT *
FROM table
WHERE <condition>

SELECT customer, spend, city
FROM sales
WHERE spend > 100
AND NOT city = 'NYC'

SELECT *
FROM table
WHERE <condition>

SELECT customer, spend, city
FROM sales
WHERE spend > 100
AND city != 'NYC'

SELECT *
FROM table
WHERE <condition>

SELECT customer, spend, city
FROM sales
WHERE spend > 100
AND city = 'NYC'
OR city = 'Amsterdam'

SELECT *
FROM table
WHERE <condition>

SELECT customer, spend, city
FROM sales
WHERE spend > 100
AND city IN ('NYC' , 'Amsterdam')

SELECT *
FROM table
WHERE <condition>

SELECT customer, spend, city
FROM sales
WHERE spend > 100
AND city NOT IN ('London', 'Paris')

SELECT *
FROM table
WHERE <condition>

SELECT customer, spend, date
FROM sales
WHERE spend > 100
AND date LIKE '2016-07-%'

SELECT *
FROM table
WHERE <condition>

SELECT customer, spend, date
FROM sales
WHERE spend > 100
AND date LIKE '20__-06-16'

SELECT *
FROM table
WHERE <condition>

SELECT customer, spend, date
FROM sales
WHERE spend > 100
AND customer LIKE '[abc]%'

SELECT *
FROM table
WHERE <condition>
ORDER BY <column>

SELECT *

FROM table

WHERE <condition>

ORDER BY <column>

SELECT customer, spend

FROM sales

WHERE spend > 100

ORDER BY spend

SELECT *

SELECT customer, spend

FROM table

FROM sales

WHERE <condition>

WHERE spend > 100

ORDER BY <column>

ORDER BY spend

What do you think will happen now?

SELECT *
FROM table
WHERE <condition>
ORDER BY <column> DESC

SELECT *
FROM table
WHERE <condition>
ORDER BY <column> [DESC | ASC]

SELECT *
FROM table
WHERE <condition>
ORDER BY <column> [DESC | ASC]
LIMIT <number>

SELECT *
FROM table
WHERE <condition>
ORDER BY <column> [DESC | ASC]
LIMIT <number>

Let's practice!

github.com/rubennaeff/sql_bootcamp

AGGREGATIONS AND GROUP BY

So far, we have just retrieved information from a single table.

Often, we'd like to gain statistics about the data,
rather than the raw entries themselves.

SELECT *
FROM table

*Select all columns
from this table*

SELECT Count(*)
FROM table

*Count all rows (containing all columns)
from this table*

SELECT Count(*)
FROM table

*Count all rows (containing all columns)
from this table*

Same as:

SELECT Count(col1)
FROM table

*Count all rows (containing 1st column)
from this table*

```
SELECT class, name, gender, age  
FROM students
```

Print class, name, gender and age of each student.


```
SELECT DISTINCT class  
FROM students
```

Only print the (unique) class names.

```
SELECT Count(DISTINCT class)  
FROM students
```

Count the number of (unique) classes.

```
SELECT Avg(age)  
FROM students
```

Print average age of all students.

```
SELECT Avg(age) AS “Average age”  
FROM students
```

Write “Average Age” as column header, instead of “Avg(age)”

```
SELECT Avg(age) AS “Average age”  
FROM students
```

Can we also print average age by gender?

```
SELECT gender, Avg(age) AS “Average age”  
FROM students  
GROUP BY gender
```

Can we also print average age by gender? Yes!

```
SELECT gender, Avg(age) AS “Average age”  
FROM students  
GROUP BY gender
```

There are usually a few common built-in operations:
SUM, AVG, MIN, MAX, COUNT

```
SELECT class, gender, Avg(age) AS “Average age”  
FROM students  
GROUP BY class, gender
```

We can also group by multiple columns.


```
SELECT class, gender, Avg(age) AS “Average age”  
FROM students  
GROUP BY 1, 2
```

Note the convenient shorthand notation!

```
SELECT class, gender, Avg(age) AS “Average age”  
FROM students  
GROUP BY 1, 2  
ORDER BY 3
```

Note the convenient shorthand notation!

```
SELECT class, Avg(age) AS “Average age”  
FROM students  
GROUP BY class
```

Suppose we'd like to only display classes
where the average age is at least 18 years.



```
SELECT class, Avg(age) AS "Average age"  
FROM students  
GROUP BY class  
WHERE Avg(age) >= 18
```

Suppose we'd like to only display classes
where the average age is at least 18 years.



```
SELECT class, Avg(age) AS "Average age"  
FROM students  
GROUP BY class  
WHERE "Average age" >= 18
```

Suppose we'd like to only display classes
where the average age is at least 18 years.



```
SELECT class, Avg(age) AS "Average age"  
FROM students  
GROUP BY class  
WHERE "Average age" >= 18
```

Suppose we'd like to only display classes
where the average age is at least 18 years.

NOTE

The WHERE clause
only accepts column
names, and not
aggregates.

It comes *before* the
GROUP BY clause.



```
SELECT class, Avg(age) AS "Average age"  
FROM students  
GROUP BY class  
HAVING "Average age" >= 18
```

Suppose we'd like to only display classes
where the average age is at least 18 years.

NOTE

The HAVING clause
accepts aggregates.

It comes **after** the
GROUP BY clause.



```
SELECT class, Avg(age) AS "Average age"  
FROM students  
GROUP BY class  
HAVING Avg(age) >= 18
```

Suppose we'd like to only display classes
where the average age is at least 18 years.

NOTE

The HAVING clause
accepts aggregates.

It comes **after** the
GROUP BY clause.

SELECT <columns>
FROM <table>
GROUP BY <columns>
HAVING <condition on aggregates>

General SQL structure

SELECT <columns>
FROM <table>
WHERE <condition>
GROUP BY <columns>
HAVING <condition on aggregates>
ORDER BY <columns>
LIMIT <number>

*General SQL structure
(putting it all together)*

SELECT <columns>
FROM <table>
WHERE <condition>
GROUP BY <columns>
HAVING <condition on aggregates>
ORDER BY <columns>
LIMIT <number>

*General SQL structure
(putting it all together)*

Let's practice s'more!

SQL BOOTCAMP

RELATIONAL DATABASES

A **relational database** is organized in the following manner:

- ▶ A database has **tables** which represent individual entities or objects
- ▶ Tables have a predefined **schema** - rules that tell it what columns exist and what they look like

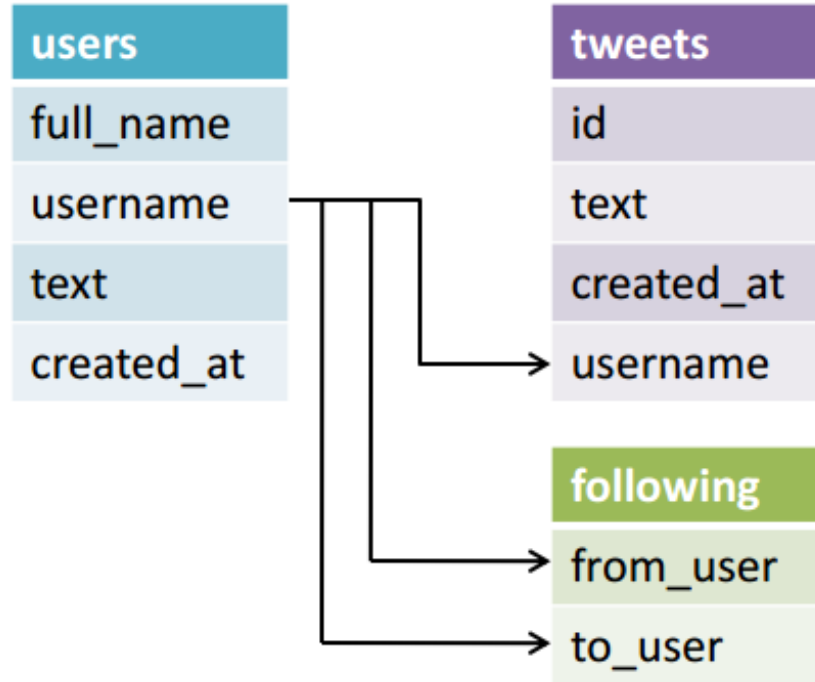
A **relational database** is organized in the following manner:

► **table**

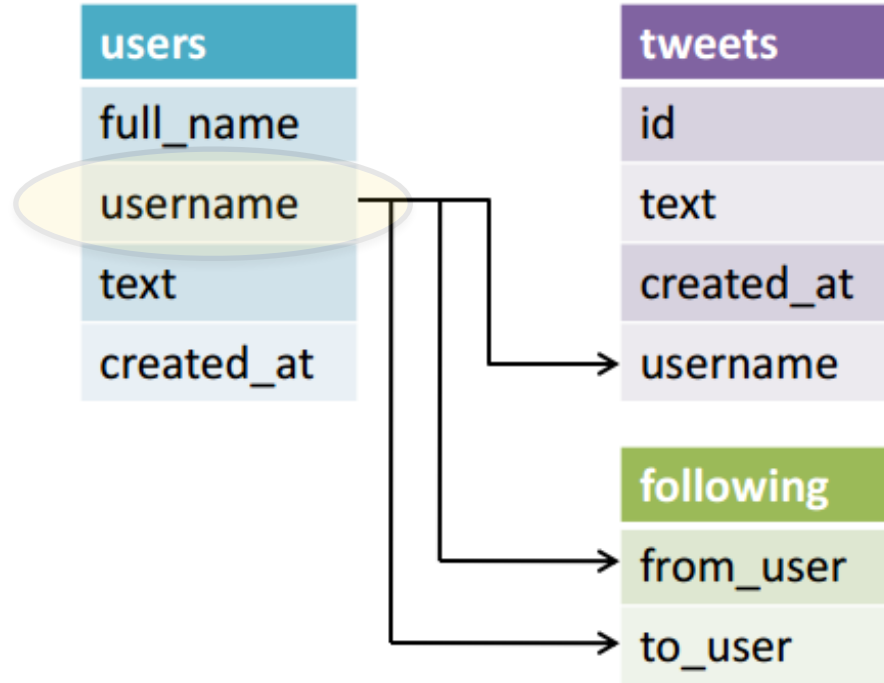
<u>id</u>	<u>first name</u>	<u>last name</u>	<u>date of birth</u>
312	Joe	Smith	1980-12-24
1532	Michelle	Anderson	1973-03-12

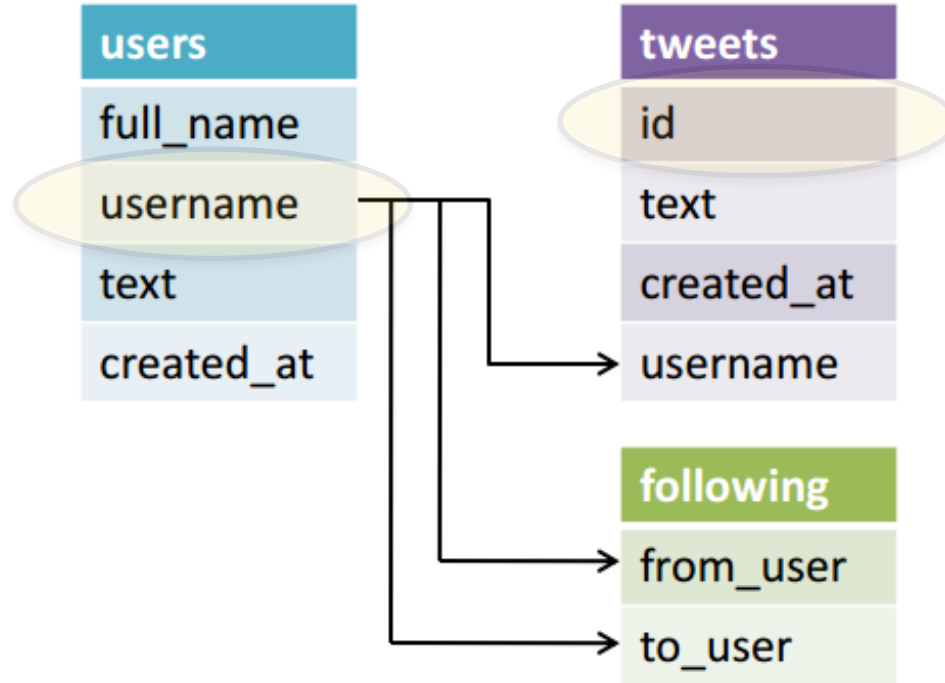
► **schema**

id	bigint
first_name	char(36)
last_name	char(36)
date_of_birth	timestamp



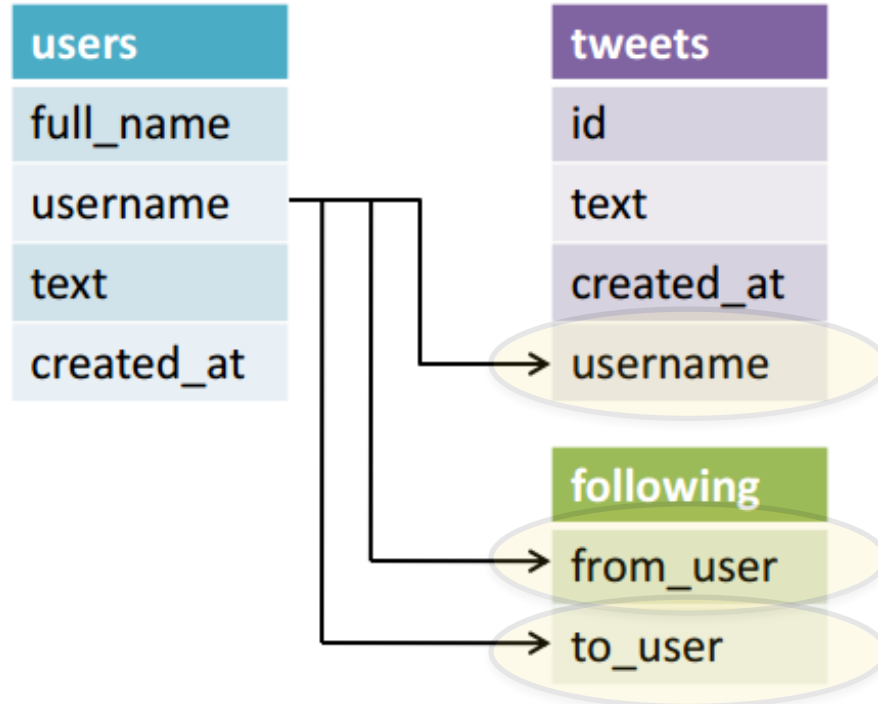
Each table should have a **primary key** column,
i.e., a unique identifier for that row





Each table should have a **primary key** column,
i.e., a unique identifier for that row

Additionally, each table can have a **foreign key** column,
i.e., an id that links this table to another



We could have had a table structure as follow:

Why is this different?

tweets
id
text
created_at
username
full_name
username
text
created_at

We could have had a table structure as follow:

Why is this different?

We would repeat the user information on each row.

This is called
denormalization

tweets
id
text
created_at
username
full_name
username
text
created_at

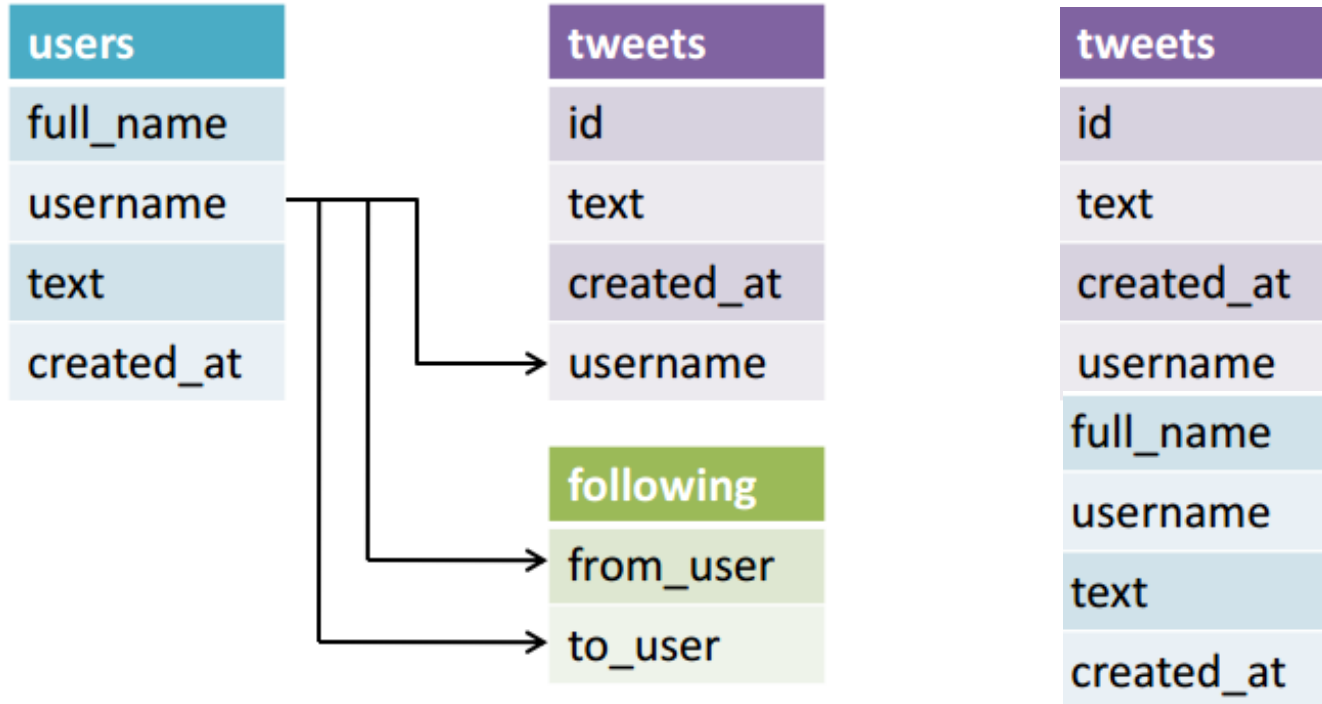
Normalized Data:

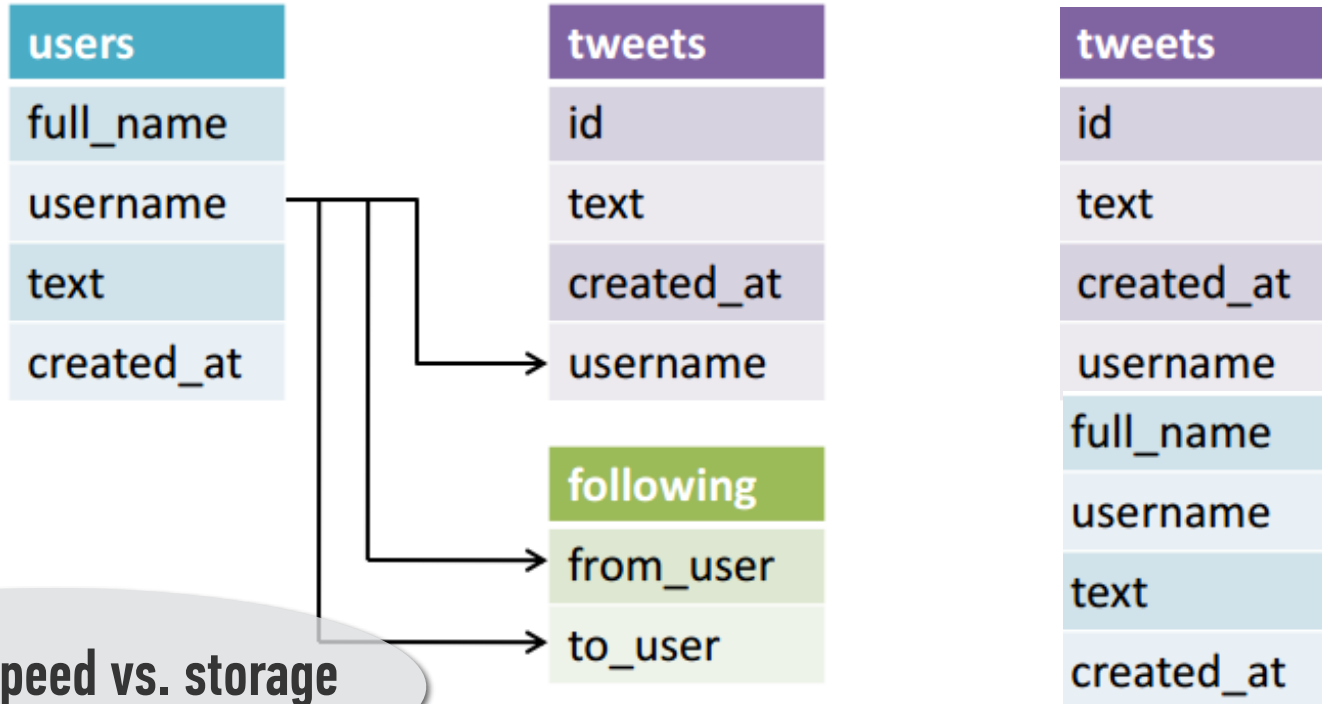
Many tables to reduce redundant or repeated data in a table

Denormalized Data:

Wide data, fields are often repeated

but removes the need to join together multiple tables





Q: How do we commonly evaluate databases?

Q: How do we commonly evaluate databases?

- ▶ *read-speed vs. write speed*

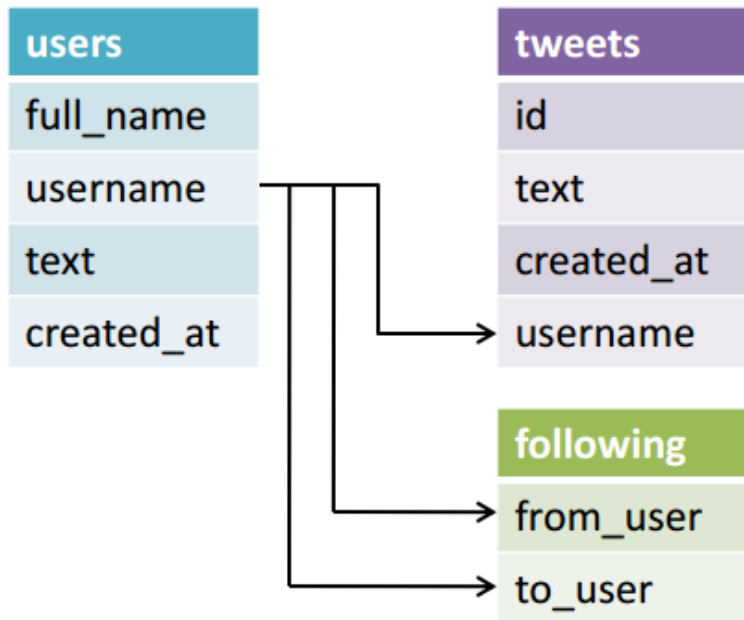
Q: How do we commonly evaluate databases?

- ▶ *read-speed vs. write speed*
- ▶ *space considerations*

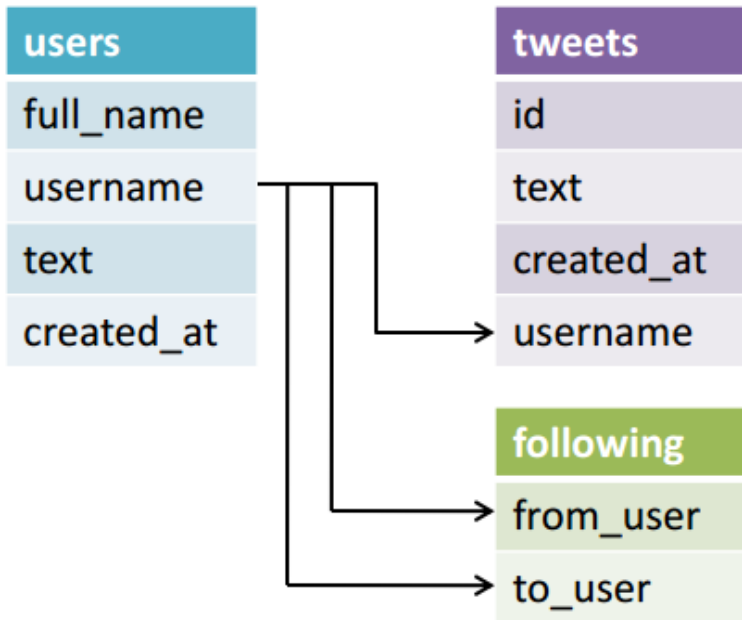
Q: How do we commonly evaluate databases?

- ▶ *read-speed vs. write speed*
- ▶ *space considerations*
- ▶ *(...and many other criteria)*

Q: Why are normalized tables (possibly) slower to read?



Q: Why are normalized tables (possibly) slower to read?



We'll have to get data from multiple tables to answer some questions

Q: Why are denormalized tables (possibly) slower to write?

tweets
id
text
created_at
username
full_name
username
text
created_at

Q: Why are denormalized tables (possibly) slower to write?

tweets
id
text
created_at
username
full_name
username
text
created_at

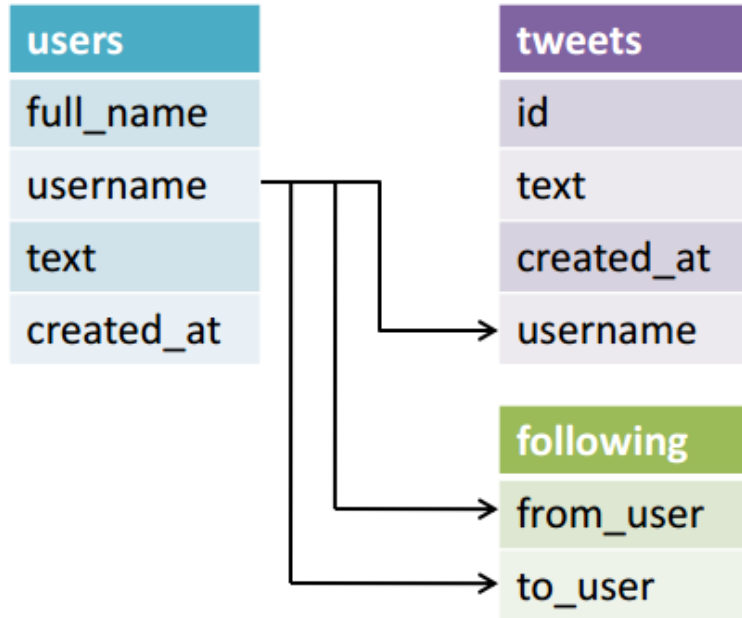
*We'll have to write more data
each time we store something*

Databases are either relational or non-relational

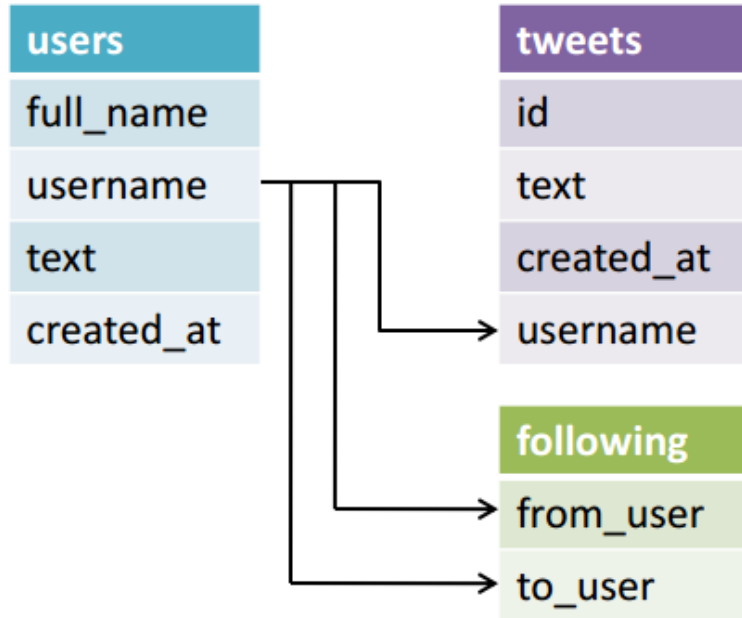
- ▶ Relational: SQL (MySQL, PostgreSQL, ...)
- ▶ Non-relational: NoSQL (MongoDB, Cassandra, ...)

SQL: THE JOIN COMMAND

Create a table with all the users' full names and their tweets

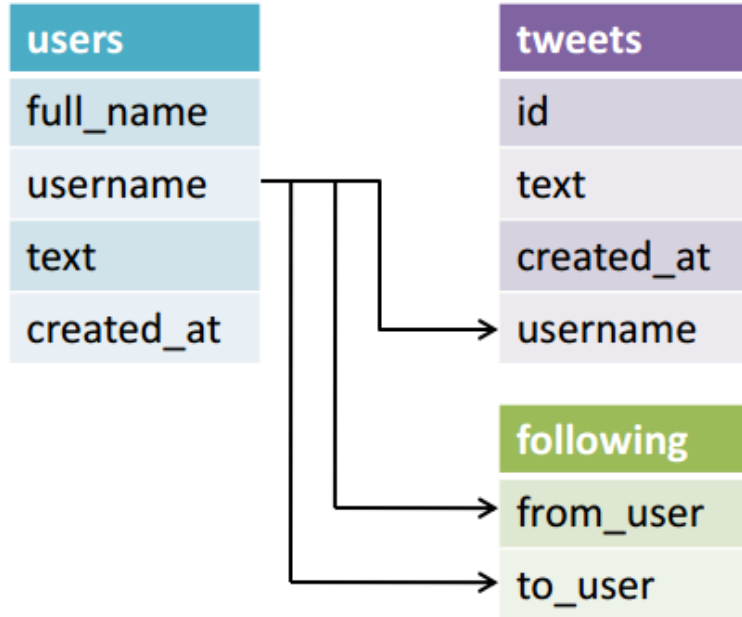


Create a table with all the users' full names and their tweets



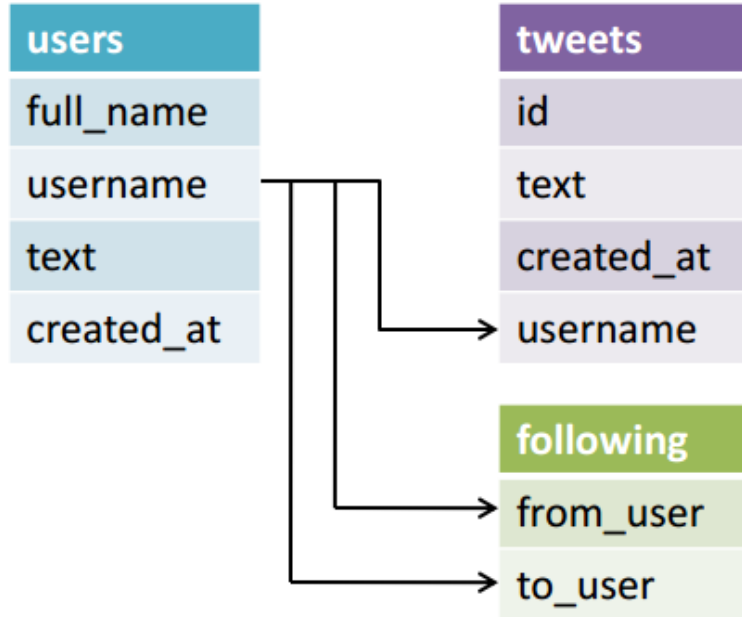
<u>full_name</u>	<u>tweet</u>
Joe Smith	Hello, world!
Joe Smith	Just tweetin'
Michelle Ng	I am eating pizza tonight

Create a table with all the users' full names and their tweets



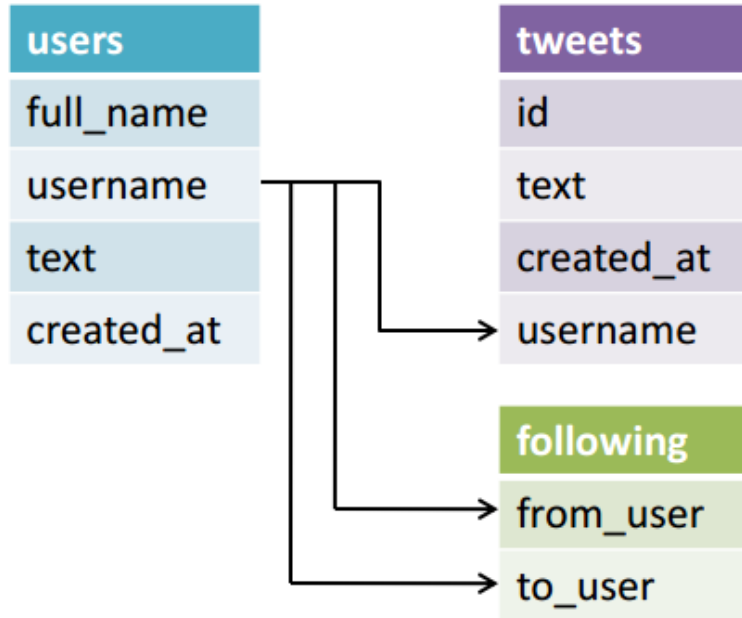
```
SELECT  
  users.full_name,  
  tweets.text
```

Create a table with all the users' full names and their tweets



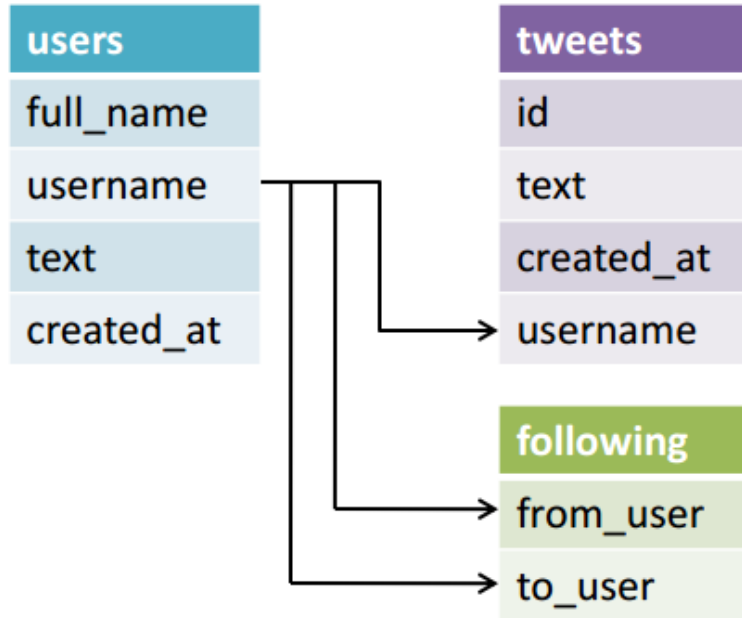
```
SELECT
  users.full_name,
  tweets.text
FROM users
```

Create a table with all the users' full names and their tweets



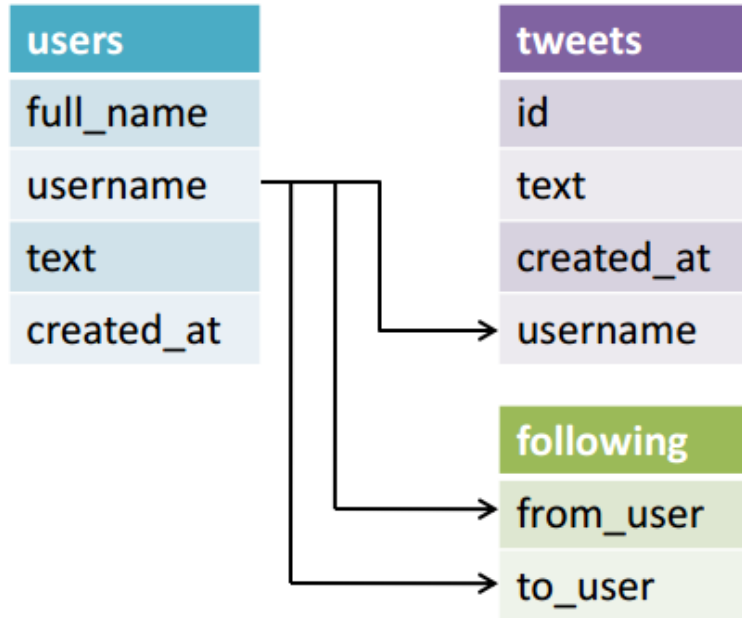
```
SELECT
  users.full_name,
  tweets.text
FROM users
JOIN tweets
```


Create a table with all the users' full names and their tweets



```
SELECT
    users.full_name,
    tweets.text
FROM users
JOIN tweets
ON users.username = tweets.username
```

Create a table with all the users' full names and their tweets



```
SELECT
  full_name,
  text
FROM users
JOIN tweets
ON users.username = tweets.username
```

SELECT <columns>
FROM <table>
JOIN <otherTable>
ON <table.key> = <otherTable.key>

General SQL structure

General SQL structure

```
SELECT <columns>  
FROM <table>  
JOIN <otherTable>  
ON <table.key> = <otherTable.key>  
JOIN <yetAnotherTable>  
ON <otherTable.key> = <yetAnotherTable.key>
```

NOTE

You can combine as many **JOINS** as you want!

General SQL structure

SELECT <columns>
FROM <table>
JOIN <otherTable>
ON <table.key> = <otherTable.key>
WHERE <condition>
GROUP BY <columns>
HAVING <condition>
ORDER BY <columns>
LIMIT <number>

General SQL structure

SELECT <columns>
FROM <table>
JOIN <otherTable>
ON <table.key> = <otherTable.key>
WHERE <condition>
GROUP BY <columns>
HAVING <condition>
ORDER BY <columns>
LIMIT <number>

NOTE

This is basically the **only query** you need to know to successfully extract data from databases.

SELECT <columns>
FROM <table>
JOIN <otherTable>
ON <table.key> = <otherTable.key>
WHERE <condition>
GROUP BY <columns>
HAVING <condition>
ORDER BY <columns>
LIMIT <number>

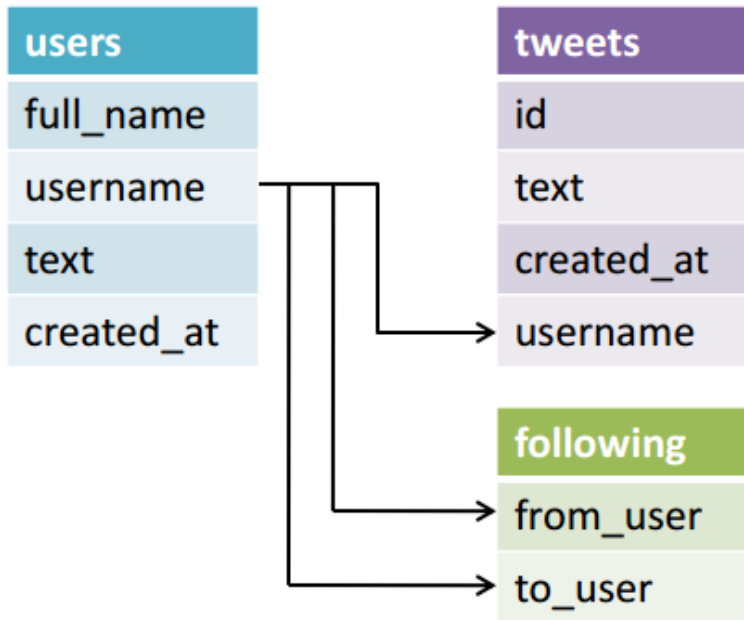
General SQL structure

Let's practice s'more!

SQL: MORE JOINS

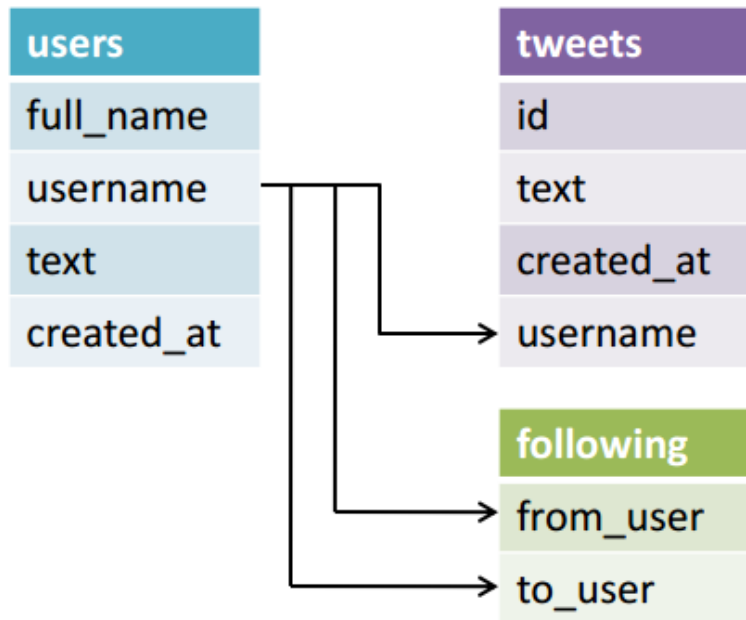
LEFT, RIGHT, INNER, OUTER

Create a table with all the users' full names and their tweets



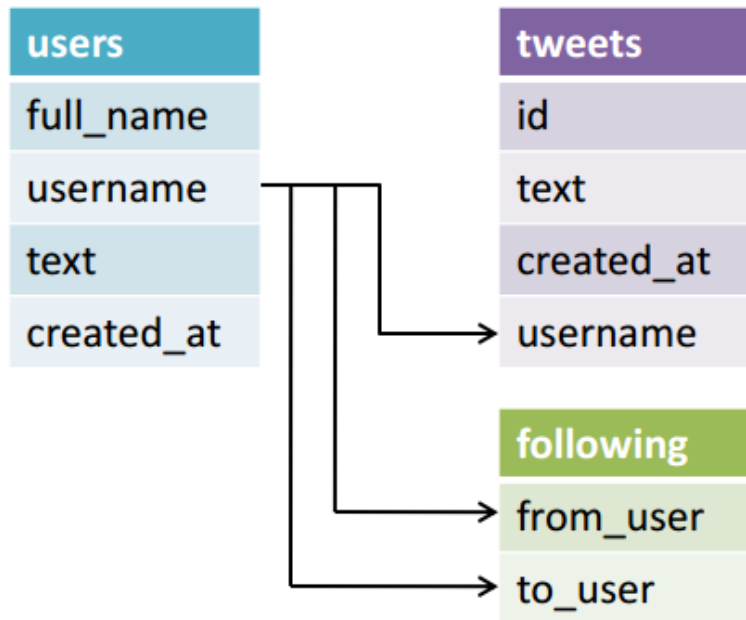
```
SELECT
  full_name,
  text
FROM users
JOIN tweets
ON users.username = tweets.username
```

Create a table with all the users' full names and their tweets



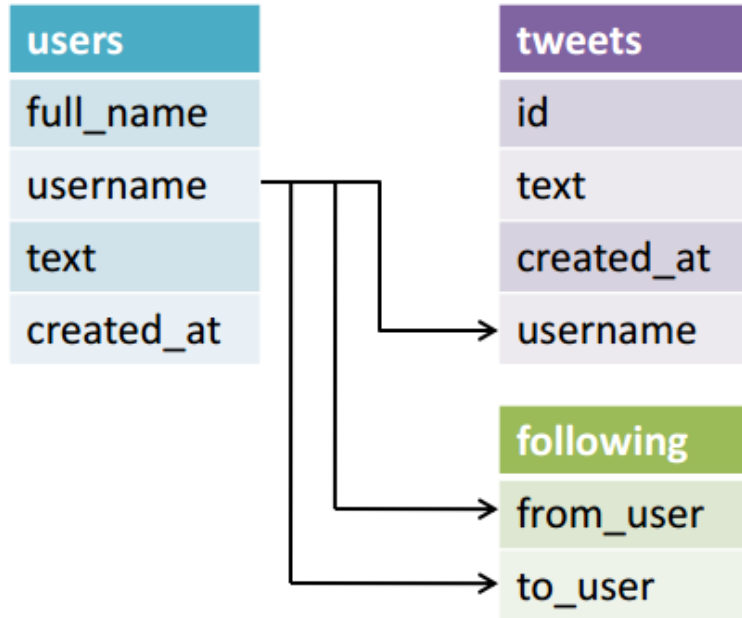
Will users who never tweeted appear in the list?

Create a table with all the users' full names and their tweets



Will users who never tweeted appear in the list? No.

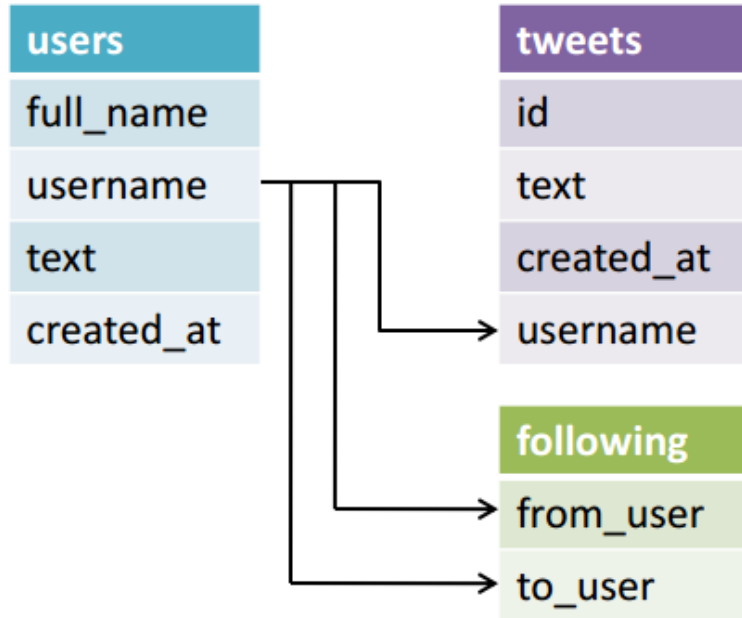
Create a table with all the users' full names and their tweets



Will users who never tweeted appear in the list? No.

Will tweets from deleted accounts still appear in the list?

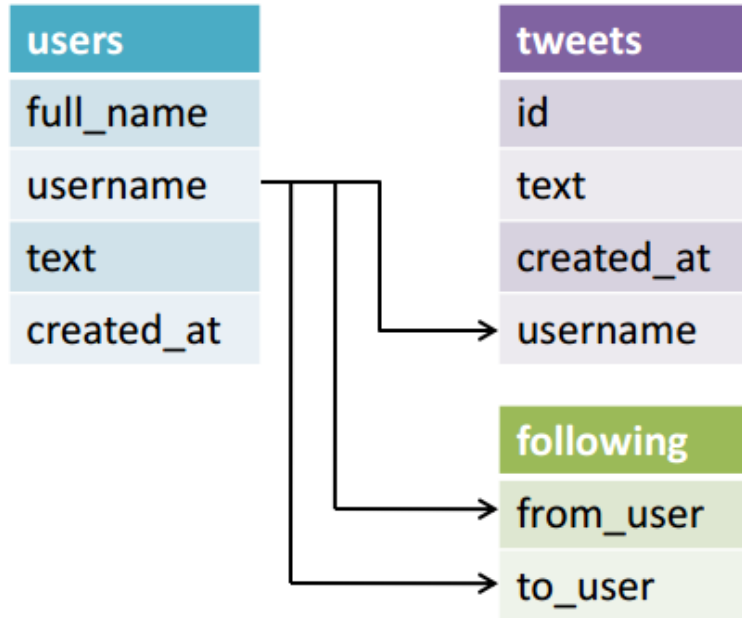
Create a table with all the users' full names and their tweets



Will users who never tweeted appear in the list? No.

Will tweets from deleted accounts still appear in the list? No.

Create a table with all the users' full names and their tweets



Will users who never tweeted appear in the list? No.

Will tweets from deleted accounts still appear in the list? No.

What if we still want them?

JOIN *will only include entries that occur in both tables.*

```
SELECT
  full_name,
  text
FROM users
JOIN tweets
ON users.username = tweets.username
```

<u>full_name</u>	<u>tweet</u>
Joe Smith	Hello, world!
Joe Smith	Just tweetin'
Michelle Ng	I am eating pizza tonight

JOIN *will only include entries that occur in both tables.*
*This is also called an **INNER JOIN**.*

```
SELECT
  full_name,
  text
FROM users
INNER JOIN tweets
ON users.username = tweets.username
```

<u>full_name</u>	<u>tweet</u>
Joe Smith	Hello, world!
Joe Smith	Just tweetin'
Michelle Ng	I am eating pizza tonight

LEFT JOIN *will always include all entries from the left table, even if there are no matches in the other table.*

```
SELECT
  full_name,
  text
FROM users
LEFT JOIN tweets
ON users.username = tweets.username
```

<u>full_name</u>	<u>tweet</u>
Joe Smith	Hello, world!
Joe Smith	Just tweetin'
Michelle Ng	I am eating pizza tonight
Jim Rogers	

RIGHT JOIN *will always include all entries from the right table, even if there are no matches in the other table.*

```
SELECT
  full_name,
  text
FROM users
RIGHT JOIN tweets
ON users.username = tweets.username
```

<u>full_name</u>	<u>tweet</u>
Joe Smith	Hello, world!
Joe Smith	Just tweetin'
Michelle Ng	I am eating pizza tonight
	OK, deleting my account

FULL OUTER JOIN *will always include all entries from both tables, even if there are no matches in the other table.*

```
SELECT
  full_name,
  text
FROM users
FULL OUTER JOIN tweets
ON users.username = tweets.username
```

<u>full_name</u>	<u>tweet</u>
Joe Smith	Hello, world!
Joe Smith	Just tweetin'
Michelle Ng	I am eating pizza tonight
Jim Rogers	OK, deleting my account

The holes in the resulting table are called **NULLs**.

NULL indicates missing data.

<u>full_name</u>	<u>tweet</u>
Joe Smith	Hello, world!
Joe Smith	Just tweetin'
Michelle Ng	I am eating pizza tonight
Jim Rogers	OK, deleting my account

The holes in the resulting table are called **NULLs**.

NULL indicates missing data.

<u>full_name</u>	<u>tweet</u>
Joe Smith	Hello, world!
Joe Smith	Just tweetin'
Michelle Ng	I am eating pizza tonight
Jim Rogers	NULL
NULL	OK, deleting my account

The holes in the resulting table are called **NULLs**.

NULL indicates missing data.

Note that **NULL** is not the same as zero or an empty string “”, it really means that there is no data.

<u>full_name</u>	<u>tweet</u>
Joe Smith	Hello, world!
Joe Smith	Just tweetin’
Michelle Ng	I am eating pizza tonight
Jim Rogers	NULL
NULL	OK, deleting my account

For example, to print a list of users without tweets, we'd write

```
SELECT full_name  
FROM users  
FULL OUTER JOIN tweets  
ON users.username = tweets.username  
WHERE tweets.text IS NULL
```



<u>full_name</u>
Jim Rogers

For example, to print a list of users without tweets, we'd write

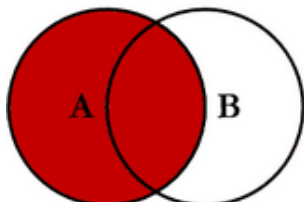
```
SELECT full_name  
FROM users  
FULL OUTER JOIN tweets  
ON users.username = tweets.username  
WHERE tweets.text IS NULL
```

full_name
Jim Rogers

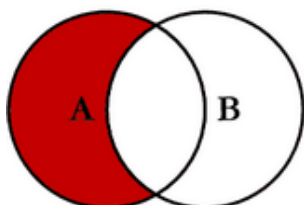
NOTE

It is common to write **IS NULL**, rather than **= NULL**, to emphasize that there is no value at all (which hence cannot be equal to anything).

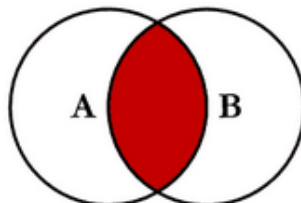
SQL JOINS



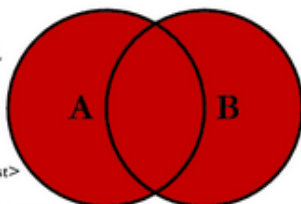
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



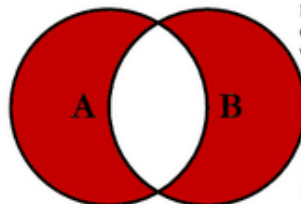
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

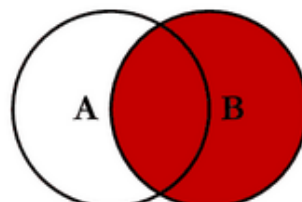


```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

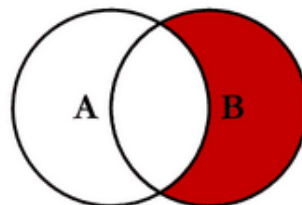


```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



SELECT <columns>
FROM <table>
[INNER|LEFT|RIGHT|FULL OUTER] JOIN <otherTable>
ON <table.key> = <otherTable.key>

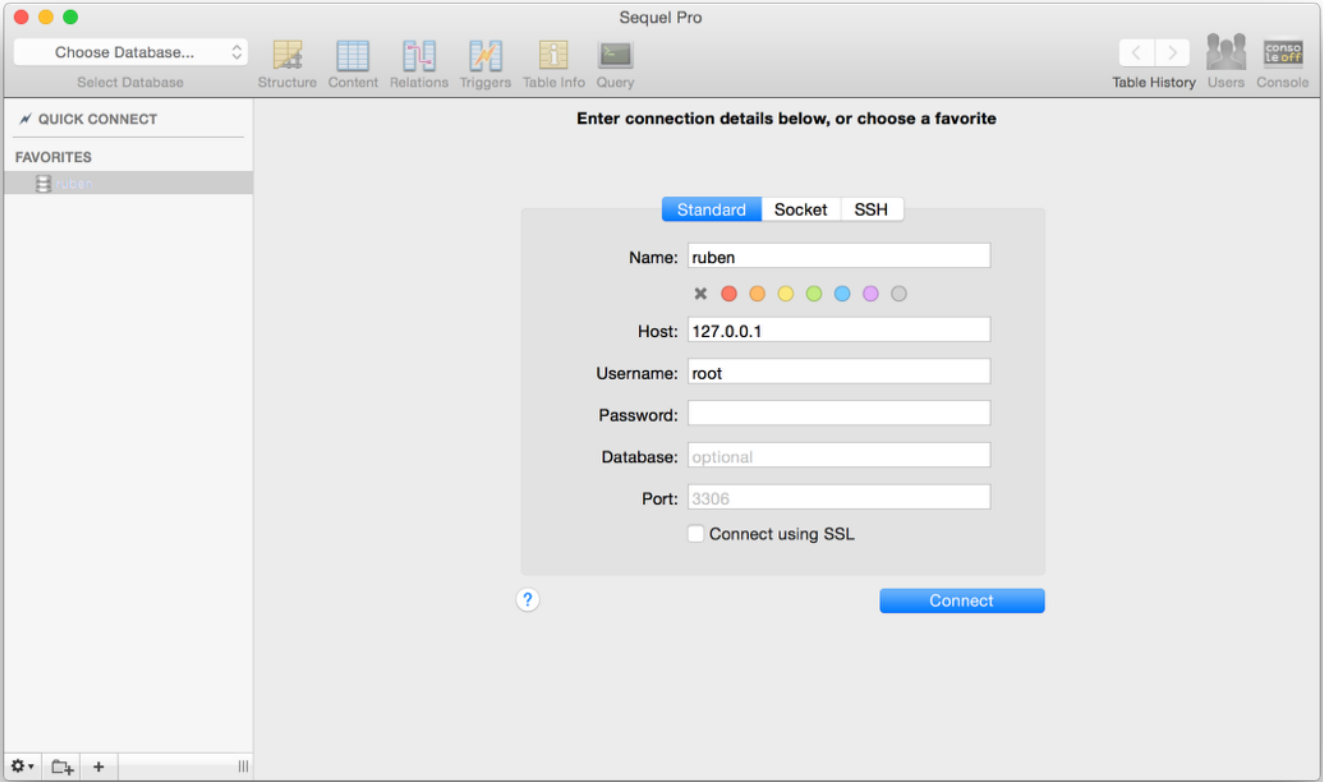
General SQL structure

General SQL structure

```
SELECT <columns>  
FROM <table>  
[INNER|LEFT|RIGHT|FULL OUTER] JOIN <otherTable>  
ON <table.key> = <otherTable.key>  
WHERE <condition>  
GROUP BY <columns>  
HAVING <condition>  
ORDER BY <columns> [DESC|ASC]  
LIMIT <number>
```

SQL BOOTCAMP

CREATING DATABASES & TABLES



CREATE DATABASE GA

CREATE DATABASE GA

DROP DATABASE GA

DROP DATABASE IF EXISTS GA


```
DROP DATABASE IF EXISTS GA;  
CREATE DATABASE GA
```

SHOW DATABASES

SQL

136**USE GA**

```
CREATE TABLE users (  
    user_id INT NOT NULL AUTO_INCREMENT,  
    first_name VARCHAR(20) NOT NULL,  
    last_name VARCHAR(30) NOT NULL,  
    age INT NOT NULL,  
    PRIMARY KEY (user_id) )
```

```
CREATE TABLE users (  
    user_id INT NOT NULL AUTO_INCREMENT,  
    first_name VARCHAR(20) NOT NULL,  
    last_name VARCHAR(30) NOT NULL,  
    age INT NOT NULL,  
    PRIMARY KEY (user_id) )
```



AUTO_INCREMENT starts
at 1 and goes up
with every record

```
CREATE TABLE users (  
    user_id INT NOT NULL AUTO_INCREMENT,  
    first_name VARCHAR(20) NOT NULL,  
    last_name VARCHAR(30) NOT NULL,  
    age INT NOT NULL,  
    PRIMARY KEY (user_id) )
```

PRIMARY KEYs are indexed
by default, must be unique,
and cannot be NULL.

BIT	0 or 1
INT	Any whole number
DECIMAL	Any number
DATETIME	A date and time
DATE	Just the date part of a datetime
CHAR(length)	Has a fixed length
VARCHAR(length)	Has a max length
...	and many more

SHOW TABLES

DESCRIBE users

DESCRIBE users

SELECT *

FROM INFORMATION_SCHEMA.COLUMNS

WHERE TABLE_NAME = 'users'

ALTER TABLE users

ADD employer_id INT

ALTER TABLE users

ADD employer_id INT

ALTER TABLE table_name

ALTER COLUMN employer_id DECIMAL

or: **MODIFY**

ALTER TABLE users

ADD employer_id INT

ALTER TABLE table_name

ALTER COLUMN employer_id DECIMAL

or: **MODIFY**

ALTER TABLE table_name

DROP COLUMN column_name

INSERT INTO users (first_name, last_name, age)

VALUES

("Bob", "Bobson", 20),

("Betty", "Bettyberg", 42)

```
DELETE FROM users  
WHERE user_id = 2
```

```
DELETE FROM users  
WHERE user_id = 2
```

```
UPDATE users  
SET first_name = 'Bobby'  
WHERE user_id = 1
```


DROP TABLE users

DROP TABLE IF EXISTS users

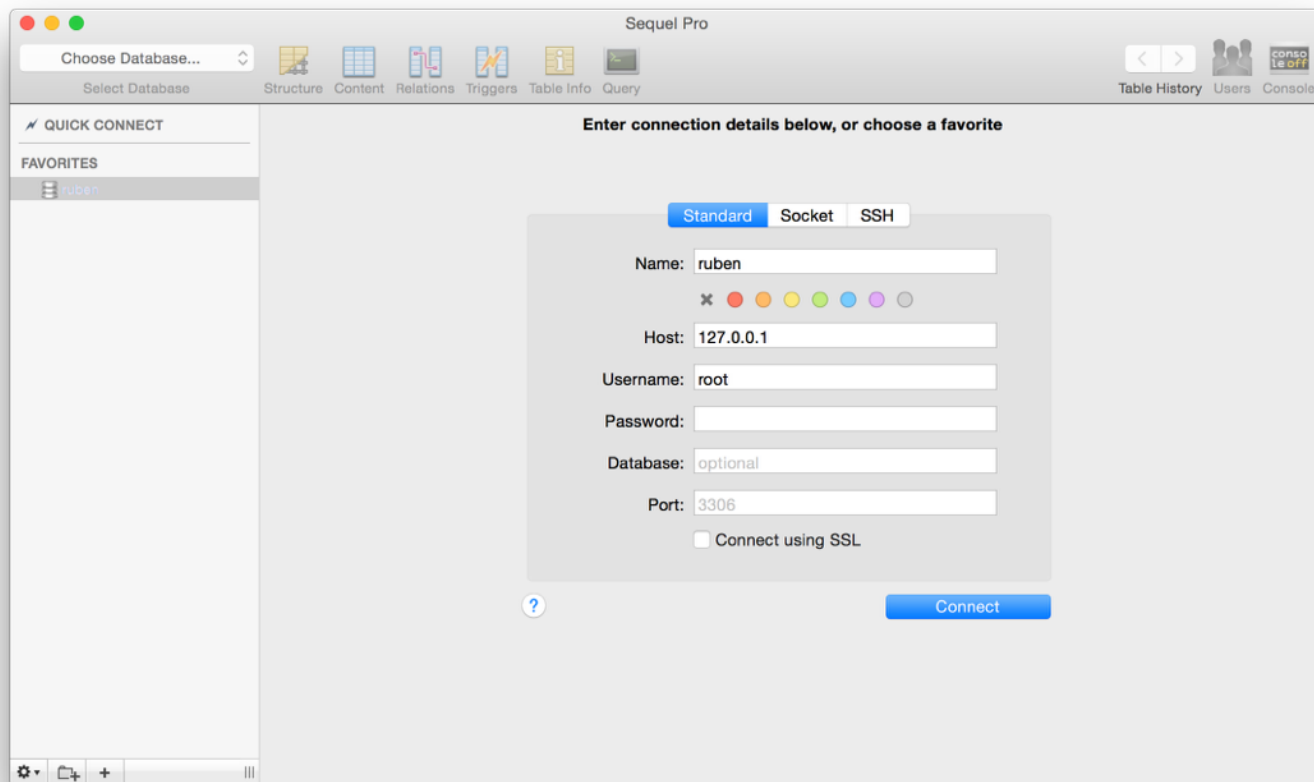
DROP TABLE users

DROP TABLE IF EXISTS users

Let's practice!

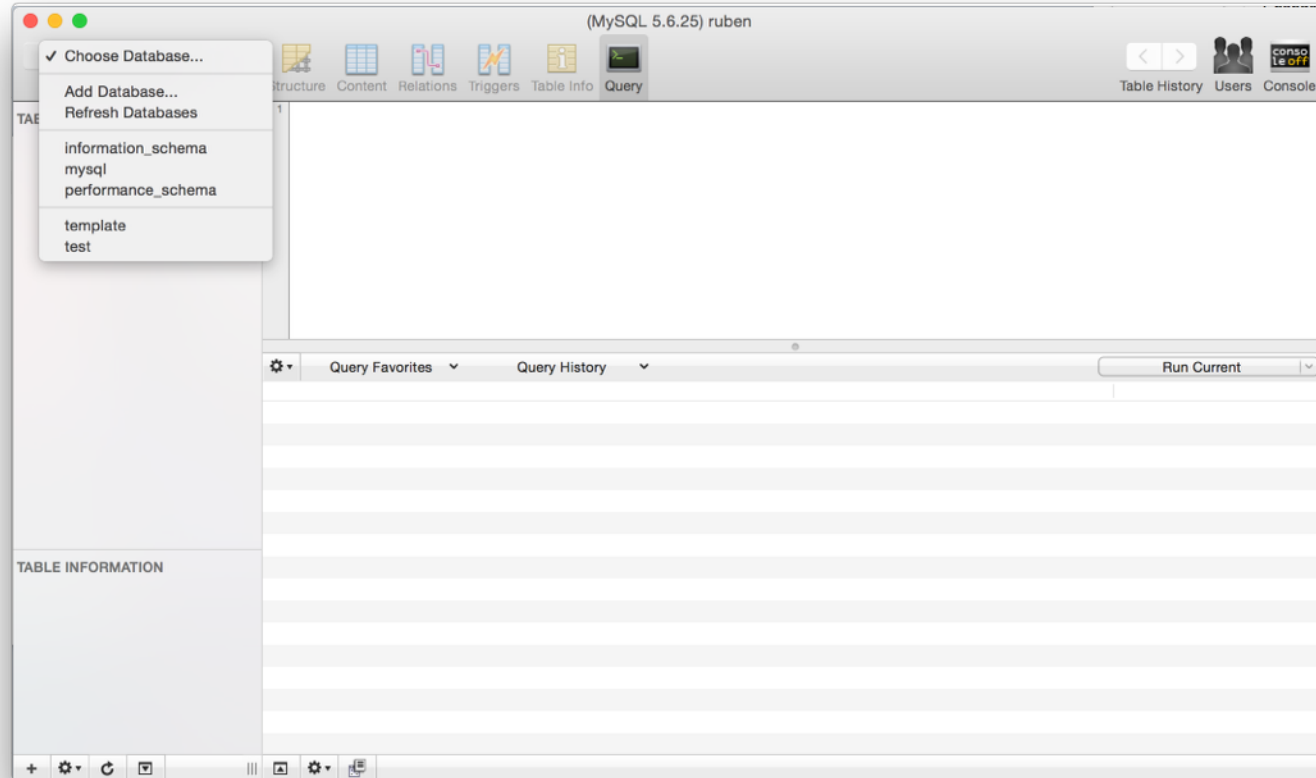
USING A FANCY CLIENT

SEQUELPRO & SQLYOG

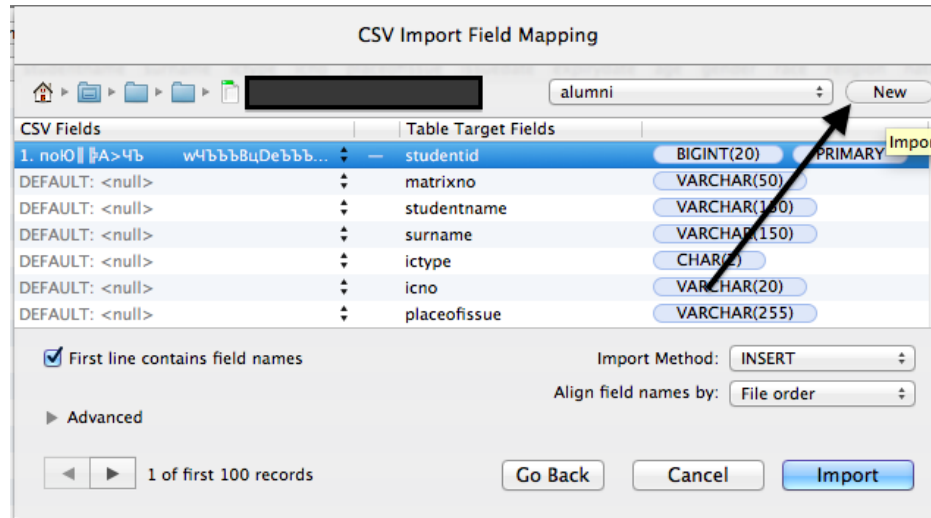


SEQUELPRO - CREATE A NEW DATABASE ("GA")

154



- ▶ Click **File | Import** and select a data file to import
- ▶ Check or uncheck box **First line contains field names**
- ▶ Click **New** if the file is a new table (not part of an existing one)



VARIOUS FUNCTIONS

SELECT 1+1

SELECT NOW()

SELECT "test"


```
SELECT CONCAT(first_name, ' ', last_name) AS full_name  
FROM users
```

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name  
FROM users
```

```
SELECT LENGTH(first_name) AS name_length  
FROM users
```

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name  
FROM users
```

```
SELECT LENGTH(first_name) AS name_length  
FROM users
```

```
SELECT LOCATE('a', first_name) AS first_a_location  
FROM users
```

SELECT

first_name,

last_name,

CASE WHEN last_name < 'n' **THEN** 'A' **ELSE** 'B' **END AS** position

FROM users

SELECT

first_name,

last_name,

IF (last_name < 'n', 'A', 'B') **AS** position

FROM users

```
SELECT *  
FROM users  
WHERE first_name IN (  
    SELECT DISTINCT first_name  
    FROM presidents  
)
```

SELECT col1, col2
FROM table

*Select some columns
from this table*

SELECT col1, col2
FROM table

*Select two columns
from this table*

SELECT col1
FROM table
UNION
SELECT col2
FROM table

*Select two columns
from this table
but list them a one column,
underneath each other*

SQL BOOTCAMP

THANK YOU