Session 5

Relational Databases (SQL I)

Previously on D@S... Relational databases are good ...

Object databases/stores

Store data in an arbitrary structure

Need arbitrary code to deconstruct object

Arbitrary code to construct new object

* Often data is recorded automatically so the initial format is somewhat arbitrary anyway and so in reality this is not really a cost.

Previously on D@S... Relational databases are good ...

Object databases/stores		Relational databases
Store data in an arbitrary structure		Potentially* need arbitrary code to deconstruct object the first time.
Need arbitrary code to deconstruct object		Store data in an a fixed "good" structure
Arbitrary code to construct new object		Simple code (8 operators) to construct new objects.

Plus automatic:
→ ability to keep data consistent when accessed by multiple users
→ ability to enforce business rules
→ ability to prevent some data errors by design

* Often data is recorded automatically so the initial format is somewhat arbitrary anyway and so in reality this is not really a cost.

Dr Georgiana Nica-Avram

Previously on D@S... Relational databases are good ...

Object databases/stores	Relational databases
Store data in an arbitrary structure	Potentially* need arbitrary code to deconstruct object the first time.
Need arbitrary code to deconstruct object	Store data in an a fixed "good" structure
Arbitrary code to construct new object	Simple code (8 operators) to construct new objects.

Purple box is where analysts spend their time

Plus automatic:
→ ability to keep data consistent when accessed by multiple users
→ ability to enforce business rules
→ ability to prevent some data errors by design

* Often data is recorded automatically so the initial format is somewhat arbitrary anyway and so in reality this is not really a cost.

Dr Georgiana Nica-Avram

The take-home message (short version)….

Normally ➡ Use relational databases

Otherwise

Too much data makes checks / writes/access to slow.

Data is always in, and processed in, objects*.

Data structure changes constantly^

➡ May store as objects

(more likely another logical structure or hopefully in a few years in "best of both world databases - week 6)

Turns out (rather than objects) **facts** make good basis (lego) pieces to construct things (i.e. cars, reports).

→ Simpler data manipulation language

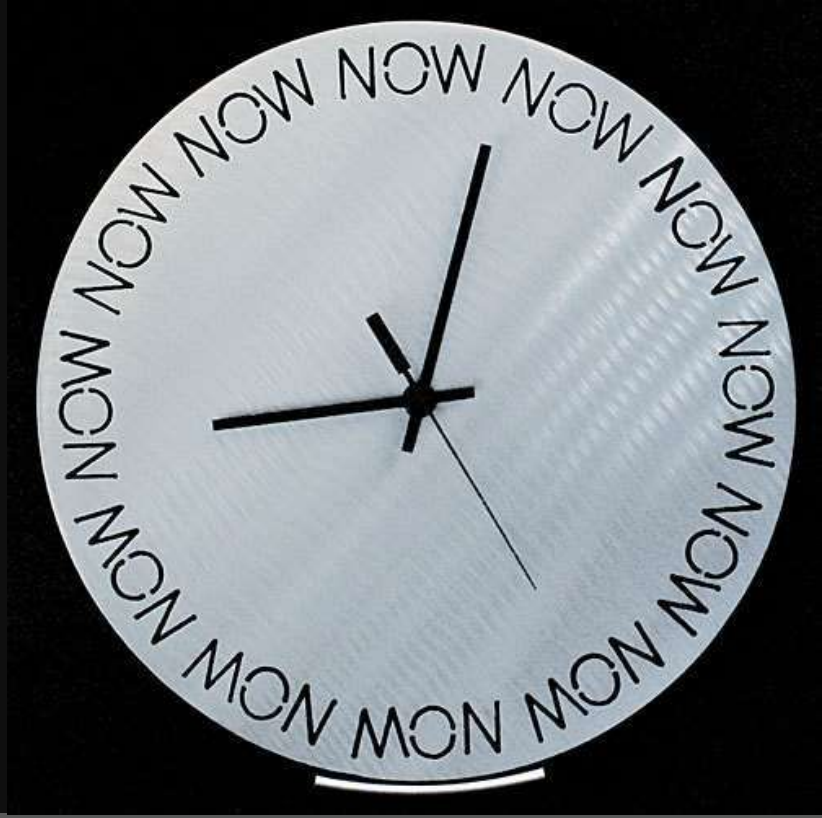
→ less time wasted
→ less human error

→ Enables constraints, business logic and consistency checks

→ leads to less error

Today...

- Understand the relational paradigm for manipulating data
- Understand and be able to use basic SQL to store and manipulate data
  - What data type would you give to each attribute?



Let's be formal with our terminology...

Entity occurrence:
a thing with distinct and independent existence

Examples: students, feet, courses, grades, ideas

HAS

Attributes:
properties of an entity

Examples: name, age, size, fees, max_mark, description

Kermit is a green Frog.
Miss Piggy is a pink pig.
Fozzy is an orange bear.

Name (Kermit) & **Colour** (Green) & **Animal** (Frog)
Name (Miss Piggy) & **Colour** (Pink) & **Animal** (Pig)
Name (Fozzy) & **Colour** (Orange) & **Animal** (Bear)



An attribute and its value is a **fact** about an entity.

Only true facts are recorded.
Some facts may uniquely identify an entity.

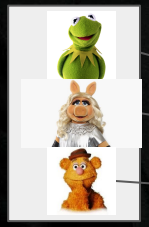
Let's be formal with our terminology...

A **table** (entity type) is a **set of entity occurrences** with the same properties.

entity 1:

entity 2:

entity 3:



name	colour	animal
Kermit	*green*	*frog*
Miss piggy	*pink*	*pig*
Fozzy	*orange*	*bear*

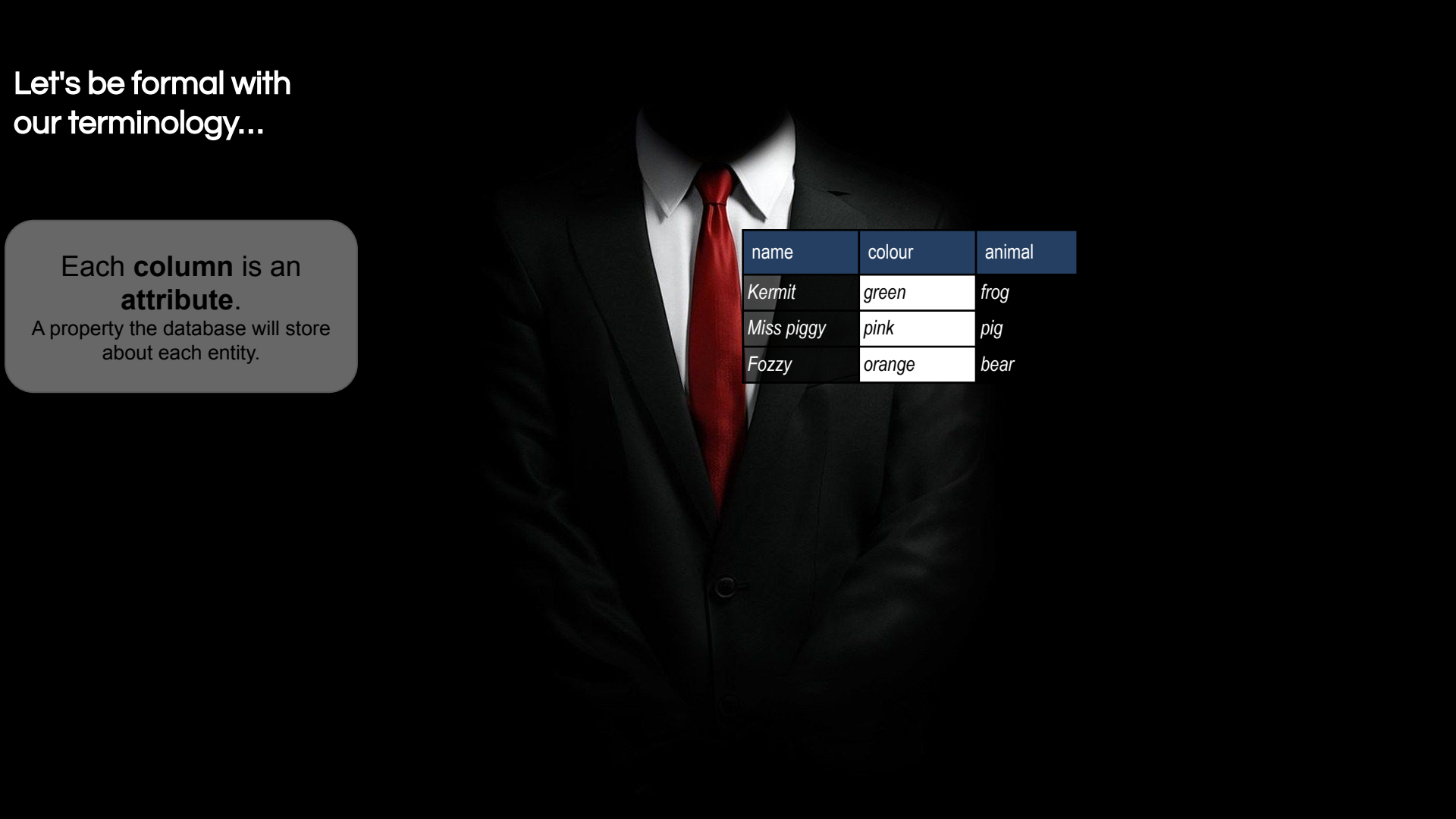
Let's be formal with our terminology...

Each **row** (tuple) in a table refers to a **distinct entity occurrence**.

name	colour	animal
Kermit	*green*	*frog*
Miss piggy	*pink*	*pig*
Fozzy	*orange*	*bear*

Let's be formal with our terminology...

Each **column** is an **attribute**.
A property the database will store about each entity.



name	colour	animal
Kermit	*green*	*frog*
Miss piggy	*pink*	*pig*
Fozzy	*orange*	*bear*

Let's be formal with our terminology...

Column (attribute) **values** for a given entity occurrence are **facts.**

If it is in the database it is a true fact.

name	colour	animal
Kermit	*green*	*frog*
Miss piggy	*pink*	*pig*
Fozzy	*orange*	*bear*

Let's be formal with our terminology...

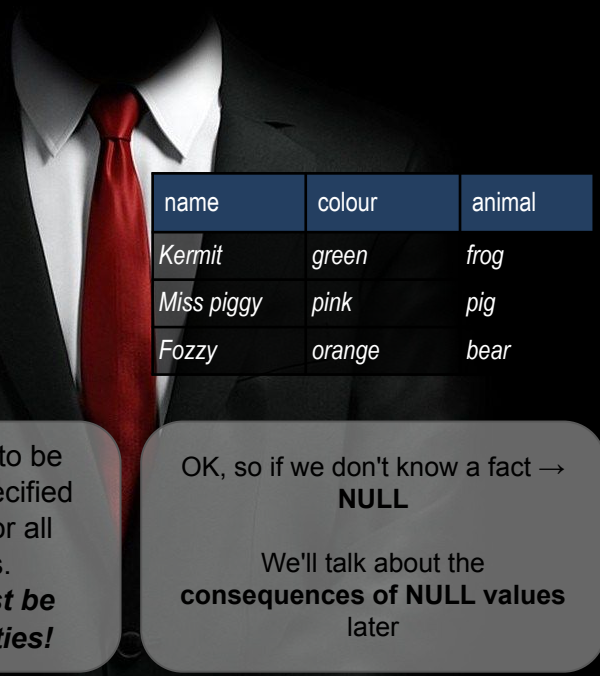
Rows are therefore sets of **jointly true facts** about entities occurrences.

name	colour	animal
Kermit	*green*	*frog*
Miss piggy	*pink*	*pig*
Fozzy	*orange*	*bear*

Let's be formal with our terminology...

Tables are **collections of entities occurrences** (rows).

Rows are **collections of facts,** where a fact is a **property** (attribute) **value pair of an entity occurrence**.



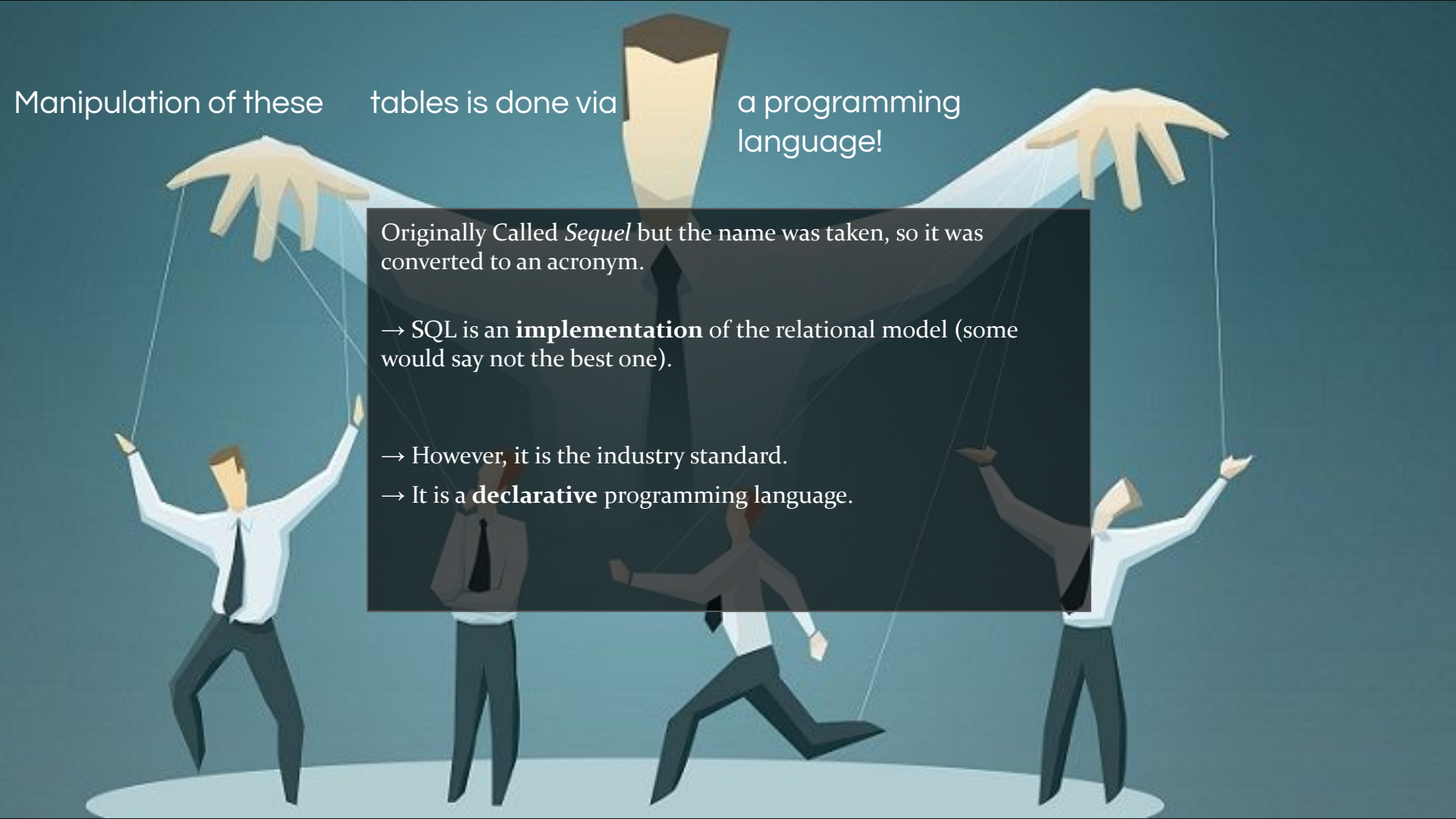
name	colour	animal
Kermit	*green*	*frog*
Miss piggy	*pink*	*pig*
Fozzy	*orange*	*bear*

Entity type properties to be stored must be pre-specified and values entered for all entity occurrences.
The same facts must be "known" for all entities!

OK, so if we don't know a fact → **NULL**

We'll talk about the **consequences of NULL values** later

Manipulation of these tables is done via a programming language!

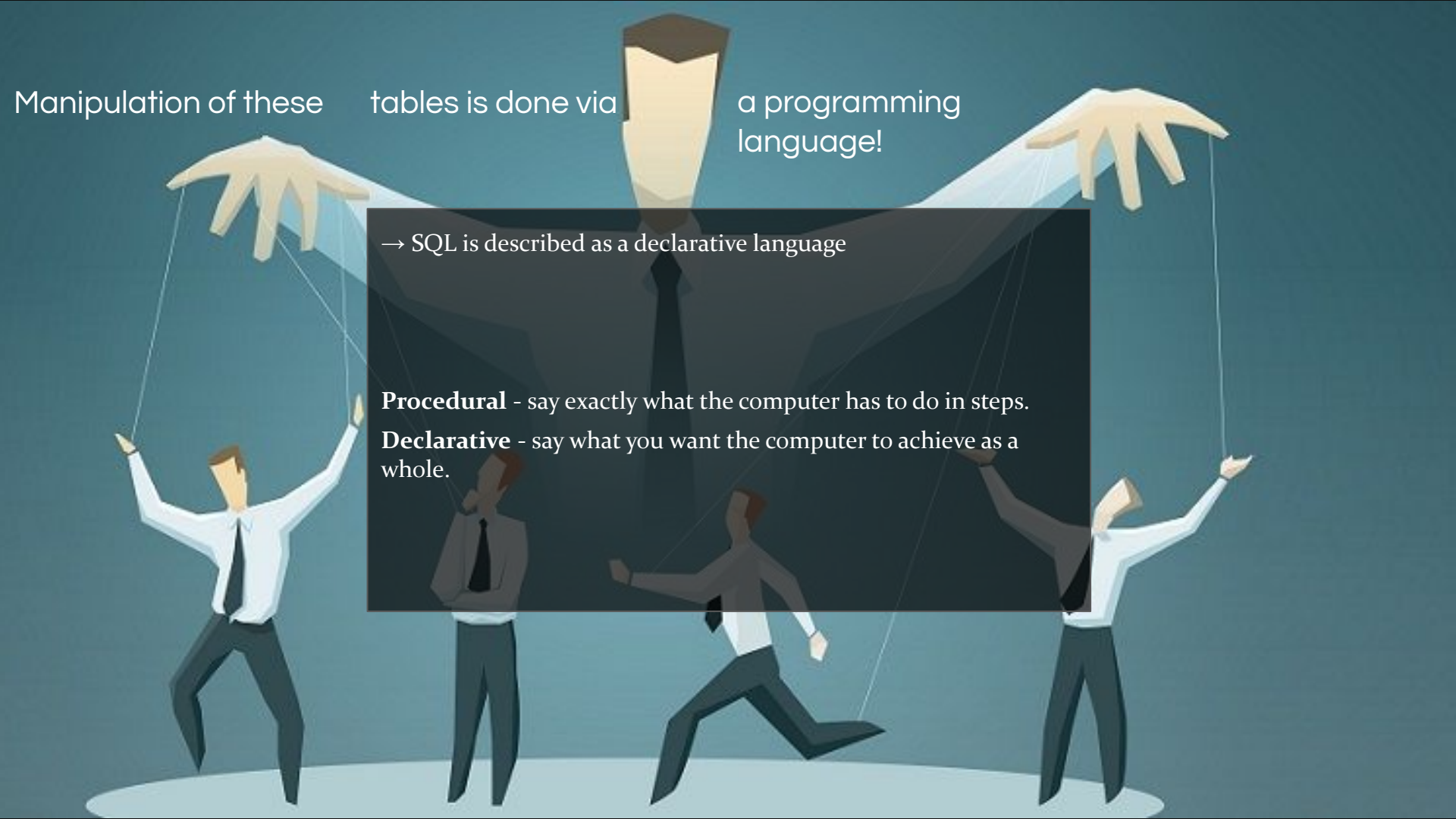


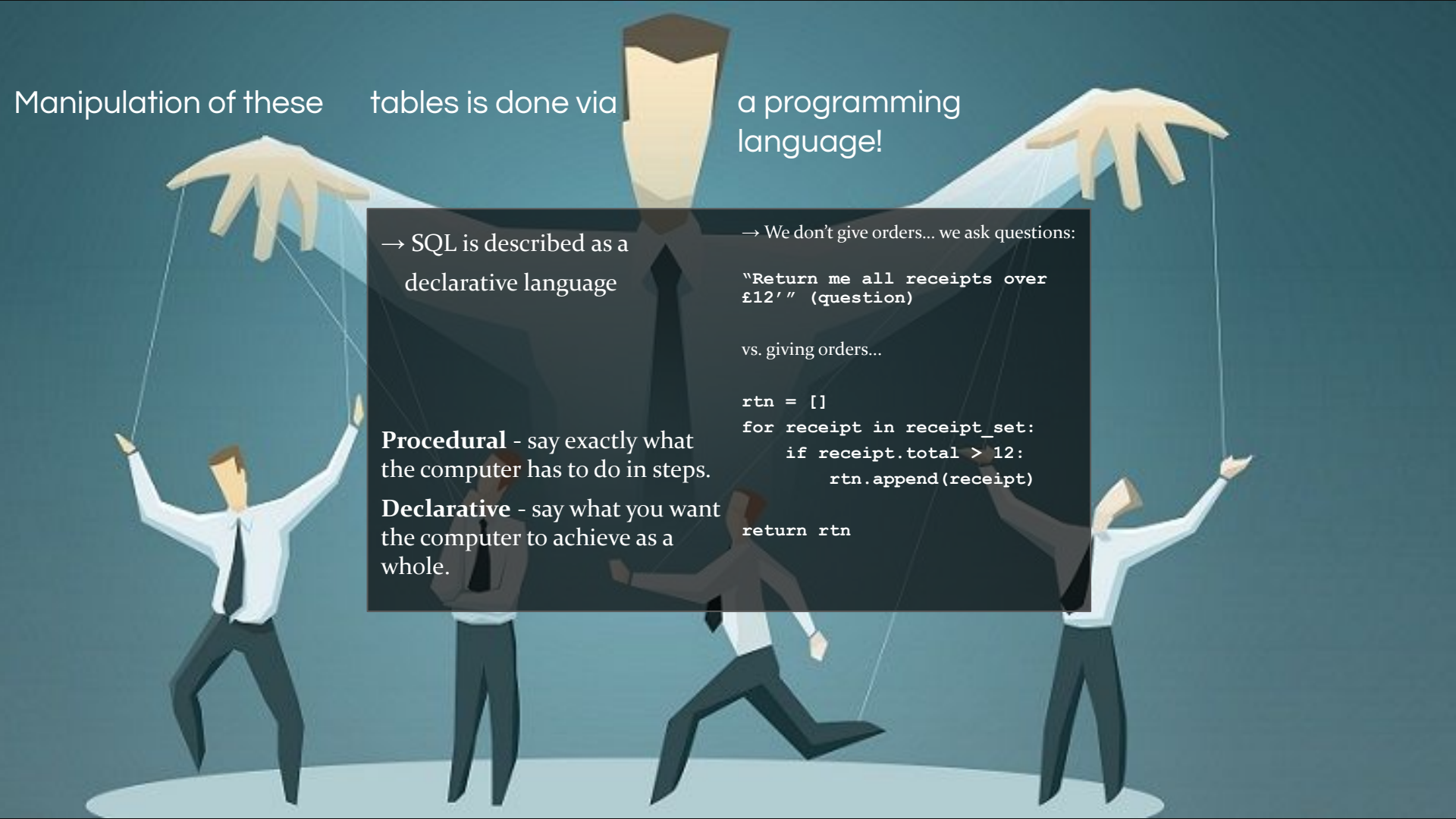
Originally Called *Sequel* but the name was taken, so it was converted to an acronym.

→ SQL is an **implementation** of the relational model (some would say not the best one).

→ However, it is the industry standard.

→ It is a **declarative** programming language.


Manipulation of these tables is done via a programming language!
→ SQL is described as a declarative language
Procedural - say exactly what the computer has to do in steps.
Declarative - say what you want the computer to achieve as a whole.


Manipulation of these tables is done via a programming language!
→ SQL is described as a declarative language
Procedural - say exactly what the computer has to do in steps.
Declarative - say what you want the computer to achieve as a whole.
→ We don't give orders... we ask questions:
"Return me all receipts over £12'" (question)
vs. giving orders...
```
rtn = []
for receipt in receipt_set:
    if receipt.total > 12:
        rtn.append(receipt)

return rtn
```

Beginning SQL...

→ SQL only has a few basic commands.

→ It is very easy to learn.

→ Designing a good database is the real challenge (coming soon!).

NOTE:

SQL is (usually) not case-sensitive, but we'll write SQL keywords in upper case for emphasis

SQL statements will be written in **BOLD COURIER FONT**

Text strings in SQL are surrounded by single quotes:

```
'I AM A STRING'
```

Single quotes within a string are doubled:

```
'I ''M A STRING'
```

The empty string: `''`

SQL: CREATE TABLE

```
CREATE TABLE       table_name
(
     column_name1  data_type,
     column_name2  data_type,
     .......
)
```

- INTEGER
- NUMERIC
- STRING
- DATE
- ...

SQL: CREATE TABLE

Optionally add
OR REPLACE

```
CREATE TABLE        table_name
(
      column_name1  data_type,
      column_name2  data_type,
      .......
)
```

- INTEGER
- NUMERIC
- STRING
- DATE
- ...

SQL: CREATE TABLE

Optionally add TEMPORARY

```
CREATE TABLE        table_name
(
      column_name1  data_type,
      column_name2  data_type,
      .......
)
```

- INTEGER
- NUMERIC
- STRING
- DATE
- ...

SQL: CREATE TABLE - Data Types

Four basic data type categories :

- Boolean
- Character based
- Number based
- Temporal types

- Plus various more complex ones...

https://spark.apache.org/docs/latest/sql-ref-datatypes.html (see SQL tab under: Supported Data Types)

Stores 2 values.

true or false

On input:
1, yes, y, t, true → true

0, no, n, false, f → false

SQL: CREATE TABLE - Data Types

Four basic data type categories :

- Boolean
- Character based
- Number based
- Temporal types

- Plus various more complex ones...

https://spark.apache.org/docs/latest/sql-ref-datatypes.html (see SQL tab under: Supported Data Types)

Three types:

- char(n)
- varchar(n)
- string

Only ever use **STRING** .

Minimal performance difference if any.

SQL: CREATE TABLE - Data Types

Four basic data type categories :

- Boolean
- Character based
- Number based
- Temporal types

- Plus various more complex ones...

https://spark.apache.org/docs/latest/sql-ref-datatypes.html (see SQL tab under: Supported Data Types)

Two types:

- Integer
- Floating-point

SQL: CREATE TABLE - Data Types

Four basic data type categories :

- Boolean
- Character based
- Number based
- Temporal types

- Plus various more complex ones...

https://spark.apache.org/docs/latest/sql-ref-datatypes.html (see SQL tab under: Supported Data Types)

Two sub-categories:

- Integer
- Floating-point

Three types:

- SMALLINT
- INT
- BIGINT

Generally use INT.

BIGINT if numbers being stored are > 2147483648 or less than < -2147483648.

SQL: CREATE TABLE - Data Types

Four basic data type categories :

- Boolean
- Character based
- Number based
- Temporal types

- Plus various more complex ones...

https://spark.apache.org/docs/latest/sql-ref-datatypes.html (see SQL tab under: Supported Data Types)

Two sub-categories:

- Integer
- Floating-point

Three types:

- FLOAT
- REAL
- NUMERIC

Generally use numeric.

Numeric guarantees what you enter is what you get. Trades-off performance.

REQUIRED for money data !

SQL: CREATE TABLE - Data Types

Four basic data type categories :

- Boolean
- Character based
- Number based
- Temporal types

- Plus various more complex ones...

https://spark.apache.org/docs/latest/sql-ref-datatypes.html (see SQL tab under: Supported Data Types)

Main types:

- DATE
- TIME
- TIMESTAMP
- TIMESTAMP_NTZ
- INTERVAL DAY

TIMESTAMP includes the time zone, NTZ indicates "No Time Zone".

Temporal types have special functions and operators (we'll learn the main ones as we go):

https://spark.apache.org/docs/2.3.0/api/sql/index.html

SQL: CREATE TABLE - Data Types

Four basic data type categories :

- Boolean
- Character based
- Number based
- Temporal types

- Plus various more complex ones...

https://spark.apache.org/docs/latest/sql-ref-datatypes.html (see SQL tab under: Supported Data Types)

Summary :

Boolean → use **BOOLEAN**

Characters → use **STRING**

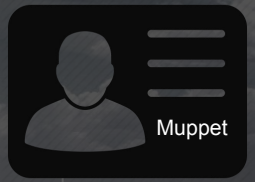
Numbers → use **INTEGER** or **NUMERIC**

Temporal → use

- **DATE**
- **TIMESTAMP**

SQL: CREATE TABLE

```
CREATE TABLE muppets(
    name STRING,
    colour STRING,
    animal STRING
    Age INTEGER
)
```



- Name
- Colour
- Animal
- Age

Worried?

→ Don't be. All code looks scary to start with.

→ There is a huge amount of help for databases on the web.

→ Good place to use as a reference:

http://www.w3schools.com/sql/

SQL: INSERT, UPDATE & DELETE

- **`INSERT`** - add a row to a table
- **`UPDATE`** - change row(s) in a table
- **`DELETE`** - remove row(s) from a table

- **`UPDATE`** and **`DELETE`** use '**`WHERE`** clauses' to specify which rows to change or remove
- BE CAREFUL with these - an incorrect **`WHERE`** clause can destroy lots of data

SQL: INSERT

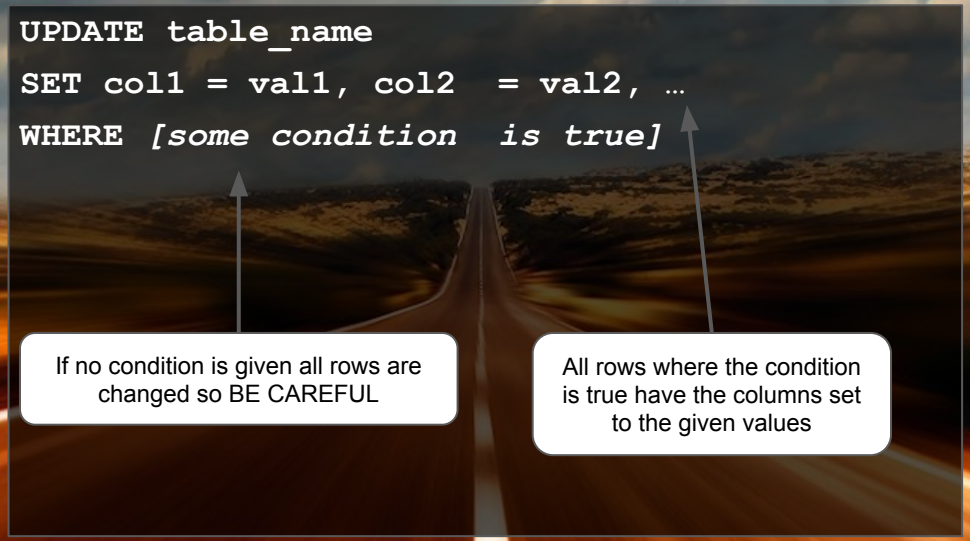
```
INSERT INTO table_name (col1, col2, …)
VALUES(val1, val2, …)
```

If you're adding a value to every column, you don't have to list them

The number of columns and values must be the same

```
INSERT INTO muppets
VALUES('Peter Andre', 'pink', 'human', 52);
```

SQL: UPDATE


UPDATE table_name
SET col1 = val1, col2 = val2, …
WHERE [some condition is true]
If no condition is given all rows are changed so BE CAREFUL
All rows where the condition is true have the columns set to the given values

SQL: UPDATE

student

id	name	year
1	John	1
2	Mark	3
3	Anne	2
4	Mary	2

```
UPDATE student
SET year = 1,
    name = 'Jane'
WHERE id = 4

UPDATE student
SET year = year + 1
```

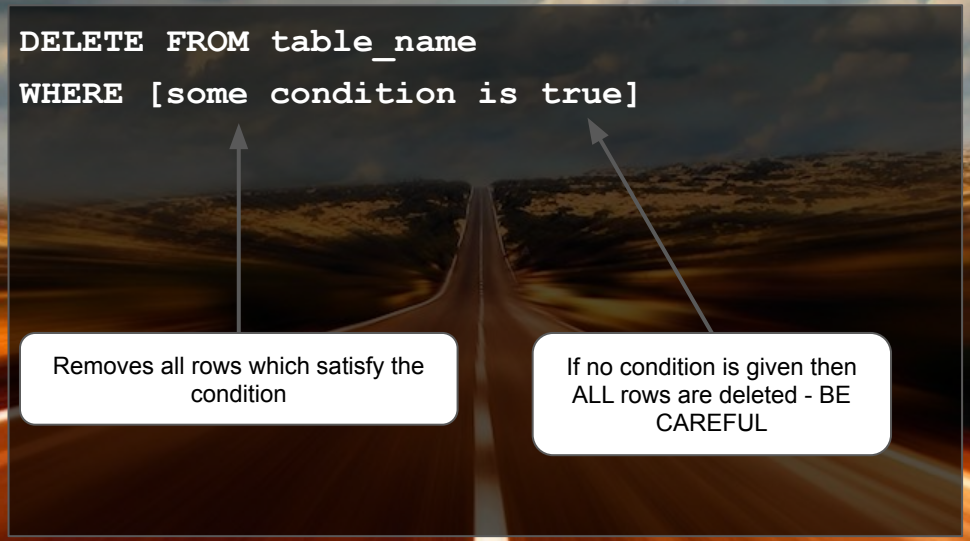
student

id	name	year
1	John	1
2	Mark	3
3	Anne	2
4	Jane	1

student

id	name	year
1	John	2
2	Mark	4
3	Anne	3
4	Mary	3

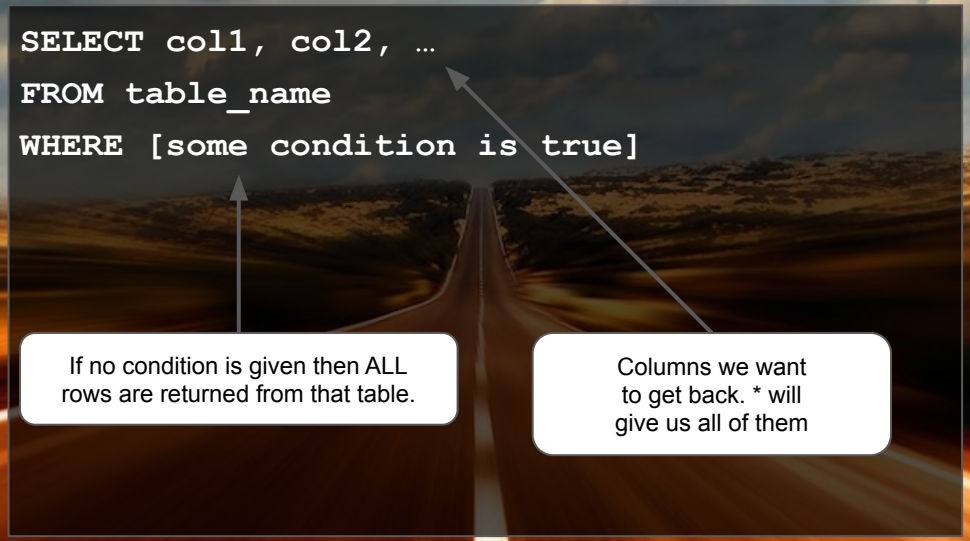
SQL: DELETE


DELETE FROM table_name
WHERE [some condition is true]
Removes all rows which satisfy the condition
If no condition is given then ALL rows are deleted - BE CAREFUL

SQL: SELECT

- **SELECT is the SQL command you will use most often**
- **Lots** of options, let's start with the basic ones!
- Worth noting that there is usually more than one way to skin a cat or do any given query.

SQL: SELECT


SELECT col1, col2, …
FROM table_name
WHERE [some condition is true]
If no condition is given then ALL rows are returned from that table.
Columns we want to get back. * will give us all of them

SQL: SELECT

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68
S104	IAI	65
S106	PR2	43
S107	PR1	76
S107	PR2	60
S107	IAI	35

student

id	first	last
S103	John	Smith
S104	Mary	Jones
S105	Jane	Brown
S106	Mark	Jones
S107	John	Brown

course

code	title
DBS	Database Systems
PR1	Programming 1
PR2	Programming 2
IAI	Intro to AI

SQL: SELECT

student

id	first	last
S103	John	Smith
S104	Mary	Jones
S105	Jane	Brown
S106	Mark	Jones
S107	John	Brown

```
SELECT id FROM student
```

id
S103
S104
S105
S106
S107

SQL: SELECT

student

id	first	last
S103	John	Smith
S104	Mary	Jones
S105	Jane	Brown
S106	Mark	Jones
S107	John	Brown

```
SELECT * FROM student
```

id	first	last
S103	John	Smith
S104	Mary	Jones
S105	Jane	Brown
S106	Mark	Jones
S107	John	Brown

SQL: WHERE

→ Usually you don't want all the rows

A WHERE clause restricts the rows that are returned

It takes the form of a condition - only those rows that satisfy the condition are returned

Example conditions:

```
mark < 40

first = 'John'

first <> 'John'

first = last

first = 'John' AND
last = 'Smith'

(mark < 40) OR (mark > 70)
```

SQL: WHERE

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68
S104	IAI	65
S106	PR2	43
S107	PR1	76
S107	PR2	60
S107	IAI	35

```
SELECT * FROM grade
 WHERE mark >= 60
```

id	code	mark
S103	DBS	72
S104	PR1	68
S104	IAI	65
S107	PR1	76
S107	PR2	60

SQL: WHERE

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68
S104	IAI	65
S106	PR2	43
S107	PR1	76
S107	PR2	60
S107	IAI	35

```
SELECT * FROM grade
 WHERE mark >= 60
```

id	code	mark
S103	DBS	72
S104	PR1	68
S104	IAI	65
S107	PR1	76
S107	PR2	60

```
SELECT DISTINCT(id)
  FROM grade
  WHERE mark >= 60
```

id
S103
S104
S107

SQL: WHERE EXERCISE FOR YOU

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68
S104	IAI	65
S106	PR2	43
S107	PR1	76
S107	PR2	60
S107	IAI	35

Write an SQL query to find a list of the ID numbers and marks of students who have passed (scored 40 or higher) the *Intro to AI (IAI)* module

id	mark
S103	58
S104	65

HINT:

```
SELECT DISTINCT(id)
FROM grade
WHERE mark >= 60
```

id
S103
S104
S107

SQL: WHERE EXERCISE FOR YOU

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68
S104	IAI	65
S106	PR2	43
S107	PR1	76
S107	PR2	60
S107	IAI	35

Write an SQL query to find a list of the ID numbers and marks of students who have passed (scored 40 or higher) the IAI module

id	mark
S103	58
S104	65

```
SELECT id, mark FROM grade
WHERE (code = 'IAI') AND
(mark >= 40)
```

We only want the ID and Mark, not the Code

quotes around the string

We're only interested in the IAI module

We're looking for entries with pass marks

Let's take a break and recap...

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68
S104	IAI	65
S106	PR2	43
S107	PR1	76
S107	PR2	60
S107	IAI	35

- **`CREATE TABLE`** - creates a table
- **`UPDATE`** - change row(s) in a table
- **`INSERT`** - add a row to a table
- **`DELETE - remove row(s) from a table`**
- **`SELECT`** - return the specified column(s) from a table
- **`WHERE`** - filters row(s) by condition

All operate on a single table.

All return a single table.

Think in tables, entities & attributes...

Time for our morning exercises...

Tasks:

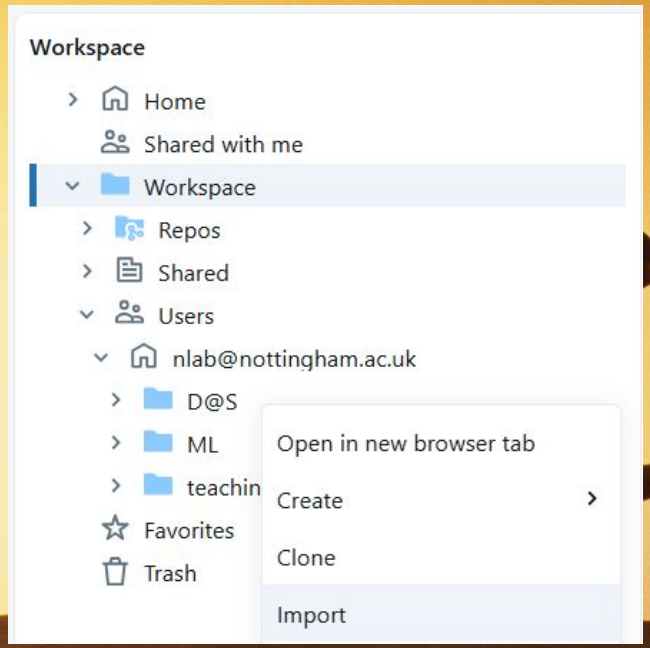
- Using pen and paper, write down the table(s) you would create to record these vehicles as digital objects in a database.
  - What attributes might you assign to each occurrence in your table(s)?
- What data type would you give to each attribute?
  - Write down the statements you would use to create and populate the table(s).

Reflect:

- What are the advantages of breaking entities into multiple table?

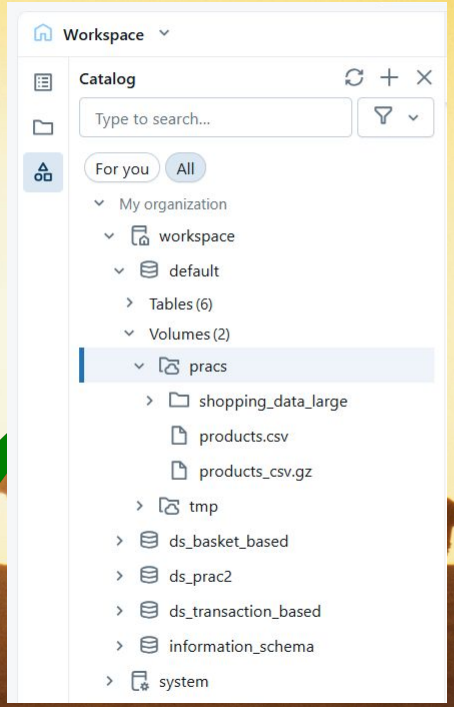
Time for our morning exercise...

To run in-class examples:



Time for our morning exercise...

Make sure you have the *products.csv* file in your volumes:



To access D@S Prac 2: